Boris Allan



GUIA DE LOGO

AMSTRAD
ESPAÑA

Guía de LOGO

Guía de LOGO

Boris Allan



GUÍA DE LOGO

Edición española de la obra

GUIDE TO LOGO
Boris Allan

publicado en castellano bajo licencia de

Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
Brentwood, Essex

Traducción:
Francisco José Izquierdo

INDESCOMP, S.A.
Avda. del Mediterráneo, 9
28007 Madrid

ISBN 84 86176 37 9
Depósito legal: M-9318-86

Impresión:
Gráficas Ema. Miguel Yuste, 27. Madrid.

Producción de la edición española:
Vector Ediciones.
Carretera de Canillas, 134.
28043 Madrid (91) 408 52 17

Contenido

Capítulo 1: Presentación de LOGO 1

Capítulo 2: Puesta en marcha de LOGO 3

2.1 Cómo cargar LOGO 3
2.2 En busca de la tortuga 5
2.3 Un poco de entrenamiento con la tortuga 7

Capítulo 3: Procedimientos en LOGO 13

3.1 Tortugas viajeras 13
3.2 Estrellas fugaces 16
3.3 Cómo definir una estrella 19
3.4 Ha nacido una estrella 21
3.5 Datos y parámetros 24
3.6 Cómo grabar en disco el trabajo realizado 26

Capítulo 4: Exploración de LOGO 29

4.1 Aprender de los errores 29
4.2 Cómo descubrir nuestros errores 30
4.3 Nombres, objetos y procedimientos 33
4.4 Otra estrella nueva 35

Capítulo 5: Programas aleatorios 37

5.1 Generación de números aleatorios 37
5.2 Una tortuga vacilante 39
5.3 Un viaje más largo 41
5.4 Cómo controlar el movimiento 42

Capítulo 6: La tortuga viajera y el color 45

6.1 Códigos de control 45
6.2 Algo más sobre los colores 47
6.3 ¿El viaje final? 48
6.4 Otros medios para mover la tortuga 49
6.5 Valores procedentes de un procedimiento 51
6.6 Textos y gráficos 52

Capítulo 7: Sonido 55

7.1 Sound 55
7.2 env 59
7.3 ent 61
7.4 release 62

Capítulo 8: Las potencias de 2 63

8.1 El juego ALTOBAJO 63
8.2 El árbol 65
8.3 Un árbol inclinado 68

Capítulo 9: Nombres y propiedades 71

9.1 Los valores de las propiedades 71
9.2 Los valores de los valores 74
9.3 Pruebas aleatorias 76

Capítulo 10: Aritmética y estadística 81

10.1 Palabras y números 81
10.2 Prioridades en aritmética 83
10.3 Combinación de listas 85
10.4 Estadística con dos variables 88
10.5 Un invento aritmético 90
10.6 Correlación y regresión 92
10.7 Sistemas estadísticos con dos variables 93

Capítulo 11: Tratamiento de matrices 95

11.1 La matriz como tabla 95
11.2 La matriz como propiedad 96
11.3 Bucles controlados 98
11.4 Cómo elegir un elemento 100
11.5 Trasposición de matrices 102
11.6 Combinación de matrices 104

Capítulo 12: PROLOGO 109

12.1 Hechos y relaciones 109
12.2 Cómo crear una base de datos 111
12.3 Cómo grabar en disco una base de datos 113
12.4 Cómo recuperar la base de datos 114

Apéndice A: LOGO: Órdenes y primitivas 117

Apéndice B: Caracteres de control de Dr. LOGO 167

Apéndice C: Discos y LOGO 169

C.1 Cómo proteger su inversión 169
C.2 Ficheros de LOGO 172
C.3 Otras funciones de interés 174

Apéndice D: Ampliaciones de LOGO 177

Capítulo 1

Presentación de LOGO

LOGO no es un lenguaje nuevo; existe desde hace bastante tiempo. Sin embargo, hace poco que está disponible para microordenadores; la versión de Digital Research, Dr LOGO, apareció en 1984.

Los ordenadores Amstrad de la serie CPC cuentan con un LOGO que es mucho más potente que cualquier versión de BASIC para microordenadores. Es posible que usted haya oído algo sobre LOGO y tenga la idea de que se trata de un sistema de gráficos para niños. En efecto, algunas versiones de LOGO no son más que un sistema de gráficos, pero no las podemos considerar como "auténticos" LOGO, sino como algo que podríamos denominar "LOGO de juguete". En cambio, Dr LOGO es un sistema potente con el que podemos llegar muy lejos.

Vamos a comenzar esta Guía de LOGO abordando los gráficos de tortuga, probablemente el aspecto más famoso de LOGO, pero nuestro objetivo final será desarrollar una base de datos con ciertas similitudes a PROLOG, un lenguaje de ordenadores de la "quinta generación". A lo largo de este camino vamos a intentar dar al lector una idea de la enorme capacidad de LOGO, un lenguaje diseñado para ampliar los horizontes del usuario.

Como cualquier libro, esta Guía ha sido escrita pensando en un lector ideal: una persona de mentalidad abierta, tal vez un programador novel, o incluso una persona que no sepa programar absolutamente nada, dispuesto a aprender algo nuevo sobre las capacidades de los ordenadores. El lector puede ser incluso un programador experto, pero lo verdaderamente importante es que esté dispuesto a leer, condición sin la cual esta Guía no serviría para nada, y a experimentar sin miedo a cometer errores.

Esta guía no va dirigida a niños pequeños; más bien está pensada para que ayude a niños un poco mayores o a los padres que tengan intención de emplear LOGO con los niños más pequeños. La mejor forma de ayudar a estos últimos a usar los ordenadores de forma constructiva consiste en comprender en primer lugar un lenguaje constructivo como es LOGO.

Tradicionalmente, se ha considerado LOGO como un lenguaje para niños pequeños, pero no es mi intención enseñar a programar a éstos, sino proporcionar la herramienta necesaria a los adultos y a niños algo mayores para que ayuden a los más

pequeños a desarrollar sus cualidades geométricas, matemáticas y lógicas. Al desarrollar este proceso, los mayores podrán explorar a fondo las técnicas de programación; mi intención es arrancar de unas pocas ideas y dejarlas extenderse dentro de los límites de la imaginación de cada uno.

Hay una filosofía a la hora de enseñar LOGO que dice que lo único que tenemos que hacer es enseñar a los niños los elementos de este lenguaje y dejarlos que jueguen con ellos. Yo no estoy de acuerdo con esta idea: deje que los niños exploren el sistema, pero un sistema potente y controlable.

Aunque hay mucha gente que se las apaña perfectamente con el teclado para controlar el ordenador, por ejemplo en los juegos, otras personas, entre las que me encuentro yo mismo, tenemos una coordinación bastante peor. La solución en estos casos consiste en emplear otro tipo de dispositivos de control, como puede ser el "joystick". Desarrollar programas que empleen los joystick es en sí un excelente ejercicio de programación, y además nos proporciona un sistema para controlar el ordenador que, en muchas ocasiones, es más fácil de usar.

La programación tiene como objeto controlar el ordenador para investigar aplicaciones de interés. Por desgracia, muchas personas emplean los ordenadores únicamente para jugar. Con LOGO podemos jugar, por supuesto, pero también podemos desarrollar programas mucho más interesantes y estimulantes desde el punto de vista intelectual.

Si vamos un poco más allá del terreno infantil, podemos emplear LOGO para cosas más "serias"; por esta razón he incluido dos aplicaciones que muestran qué podemos conseguir con este lenguaje.

Antes de comenzar con Dr LOGO, quiero dar las gracias a Chris Lusby-Taylor y a Tony Harris de Digital Research por su ayuda y colaboración, y a John Cunliffe por las abundantes y fructíferas discusiones que ambos hemos mantenido.

Capítulo 2

Puesta en marcha de LOGO

Este capítulo de introducción tiene por objeto servirle de ayuda para que aprenda rápidamente a emplear un lenguaje realmente potente: LOGO. Todo lo que usted necesita para seguir este capítulo es el disco de LOGO que acompaña a su ordenador. Nuestro objetivo principal es indicar en líneas generales cómo funciona LOGO. En principio me he limitado a las posibilidades más sencillas, referentes a los gráficos de tortuga y a la disposición de la pantalla, porque considero que ésta es la forma más rápida de comprender cómo está organizado el lenguaje.

Si el lector ya ha cargado LOGO en su ordenador, o sabe cómo hacerlo, puede saltarse esta sección; nos encontraremos en el próximo apartado, que se titula "En busca de la Tortuga".

2.1 Cómo cargar LOGO

En primer lugar, lea las instrucciones referentes al manejo de discos en el manual del ordenador. A continuación tome el disco de CP/M que contiene LOGO en su cara 2 e introdúzcalo en la unidad de discos. Una vez aparezca la señal de BASIC **Ready**, aparece también el cursor, que indica que el ordenador está esperando que le dé una orden. Para comenzar su exploración por el mundo de LOGO, escriba:

¦cpm [INTRO]

con lo que se borra la pantalla y oirá que la unidad de disco se pone en funcionamiento. [INTRO] significa que hay que pulsar la tecla INTRO. Antes de que vuelva a cambiar la pantalla verá durante un momento:

CP/M 2.2 – Amstrad Consumer Electronics plc

A>LOGO

en un ancho de 80 columnas por línea. A continuación se borra la pantalla y aparece el mensaje **Amstrad LOGO V1.1**. La pantalla vuelve a cambiar rápidamente y se encontrará con una interrogación

?

en el ángulo izquierdo, seguida por el cursor. Hasta aquí no debería haber tenido ningún problema; pero en caso de que, por ejemplo, el resultado de

¦cpm

sea:

CP/M 2.2 – Amstrad Consumer Electronics plc

A>

y no se cargue LOGO, lo más probable es que haya empleado un disco equivocado (o un disco correcto pero colocado al revés). También puede comprobar el contenido del disco por medio de la orden **DIR**. Si las cosas no van bien y no ha conseguido cargar LOGO, espere un momento; en caso contrario escriba:

?bye [INTRO]

con lo que volvemos al modo de 80 columnas, y aparece el mensaje:

A>

Esto quiere decir que está en CP/M.

Si usted no consiguió cargar LOGO y el ordenador le mostró este mensaje, ya sabe que se encontraba en CP/M. Por supuesto, si el mensaje fue otro, como por ejemplo

Drive A: disc missing
Retry, Ignore or Cancel?

quiere decir que no había puesto ningún disco en la unidad. El ordenador cuenta con otros mensajes que pueden informarle de que el disco no contiene CP/M o de que no está en buenas condiciones. En caso de duda o confusión consulte el manual de CP/M.

Si estando en CP/M da la siguiente orden:

A>dir [INTRO]

puede comprobar que obtiene la respuesta:

A: LOGO COM: SETUP COM: AMSDOS COM
A>

dado que ha solicitado un directorio de los ficheros que contiene el disco. En caso de que se haya equivocado en algo, el directorio tendrá muchos más ficheros, o tal

vez muchos menos. En el primer caso, si algunos de los ficheros son los siguientes:

MOVCPM COM: PIP COM

quiere decir que tiene el disco del revés, con CP/M hacia arriba en lugar de estar en esa posición la cara con LOGO, como debería ser.

Ahora coloque correctamente el disco, con LOGO hacia arriba (si estaba bien puesto no lo toque), y vuelva a LOGO desde CP/M. Para ello, siga estas instrucciones:

1. Si originalmente usted consiguió entrar en LOGO y salió del mismo por medio de **bye**, todo lo que debe hacer es escribir:

A>logo [INTRO]

y volverá otra vez a LOGO

2. Si hasta ahora no ha conseguido entrar en LOGO, hay que reinicializar el sistema, cosa que se consigue de forma automática por medio de la rutina SETUP. Debe volver a CP/M para reajustar algunas teclas que emplea LOGO de forma especial; para ello escriba:

A>amsdos [INTRO]

para pasar a BASIC; y desde BASIC:

¦cpm [INTRO]

al igual que antes. Es muy importante partir siempre de BASIC, aunque luego entre y salga de LOGO, porque éste es el único sistema que permite ajustar LOGO correctamente.

En caso de que se le presenten otros problemas, tendrá que pedir consejo a alguien con experiencia que pueda estudiar su caso particular, aunque siempre es interesante echar un vistazo al apéndice "Discos y LOGO", en el que podría encontrar la solución a su problema.

2.2 En busca de la tortuga

Una vez que LOGO está listo para trabajar, aparece en pantalla una interrogación:

?

y junto a ella el cursor. Veamos qué sucede al escribir las dos letras **st**

?st [INTRO]

Ante esta instrucción, que es la abreviatura de "showturtle" (mostrar tortuga), el ordenador puede reaccionar de dos formas distintas.

La primera posibilidad es la siguiente:

I don't know how to ?st
?

que significa "No sé cómo hacer ?st", dado que hemos escrito **?st** en lugar de **st**. Siempre que escribamos algo incorrecto, como por ejemplo **sh** en lugar de **st**, LOGO volverá a decirnos que no lo reconoce. Normalmente el ordenador muestra el mensaje **I dont't know how to ...** (No sé cómo hacer ...).

st es una "primitiva" de LOGO, es decir, una instrucción interna que permite realizar una secuencia de operaciones. Más adelante, cuando cree sus propias secuencias de instrucciones y les dé nombres, obtendrá lo que se conoce como un "procedimiento".

En LOGO, tanto las primitivas como los procedimientos tienen como objeto realizar alguna tarea, así que la otra respuesta posible del ordenador a su primera orden es borrar la pantalla, quedando ésta limpia en sus tres cuartas partes superiores, salvo un pequeño dibujo en el centro. Esto es lo que se denomina "pantalla gráfica". En este momento la interrogación se encuentra situada en la parte inferior de la pantalla, en el extremo izquierdo de la quinta línea contando desde abajo, junto al cursor. Por último, casi en el centro de la pantalla hay una pequeña figura.

Esta figura es lo que denominamos "tortuga".

Si examina la tortuga con detalle, verá que tiene forma de punta de flecha dirigida hacia la parte superior de la pantalla; el trazo que constituye la cola de la flecha confirma dicha dirección vertical.

Ahora escriba:

?fs [INTRO]

El mensaje que acaba de escribir desaparece. Toda la pantalla queda como estaba previamente la parte superior de la misma y, si teclea algún mensaje, no se reproduce en la pantalla.

¿Cómo puede volver a la situación en la que se ven los mensajes que escribe? Aunque no se reproduzca en pantalla, escriba **ts** [INTRO]; la pantalla se ilumina, la tortuga desaparece y volvemos a encontrarnos en el ángulo superior izquierdo el signo de interrogación y el cursor. Para volver a la situación en que la pantalla se encuentra dividida en una zona para la tortuga y una zona para el texto escriba **ss**. Fundamentalmente hay tres formas que permiten combinar en pantalla los espacios para la tortuga y para el texto. La instrucción **ts**, que viene de "Textscreen" (pantalla de textos), dedica toda la pantalla a presentación de textos; **ss**, abreviatura de "Splitscreen" (pantalla dividida), divide la pantalla dejando una zona para la tortu-

ga y otra para los textos. Por último, **fs**, "Fullscreen" (pantalla completa), reserva toda la pantalla para los gráficos de la tortuga.

2.3 Un poco de entrenamiento con la tortuga

Hasta aquí hemos visto en pantalla el texto y la tortuga, pero hasta este momento la tortuga no ha hecho nada de interés salvo aparecer y desaparecer.

Si ahora escribe:

```
?rt 30 [INTRO]
?
```

la tortuga gira un poco hacia la derecha. La sentencia **rt 30** ha hecho que la tortuga gire hacia la derecha en el ángulo que indica el número que sigue a la propia instrucción. Esta instrucción realiza una acción, produce una consecuencia; es otro ejemplo de primitiva de LOGO.

A continuación vamos a ver qué sucede cuando repetimos dos veces la primitiva anterior. Observe que el texto aparece en la parte inferior de la pantalla.

```
?rt 30 [INTRO]
?rt 30 [INTRO]
?
```

Una vez que el ordenador ha ejecutado la primera de las órdenes, el lado que inicialmente estaba situado a la izquierda pasa a estar casi horizontal. La segunda primitiva hace que la parte posterior de la tortuga quede vertical, con lo que la flecha apunta hacia la derecha.

Escriba la siguiente orden:

```
?lt 90 [INTRO]
?
```

y la tortuga queda de nuevo apuntando hacia arriba, como al principio. La figura 2.1 muestra los efectos de esta sucesión de instrucciones.

La primitiva que hace que la tortuga gire un ángulo de 90 grados hacia la izquierda, **lt**, compensa las tres órdenes previas con las que había hecho que la tortuga girase 30 grados hacia la derecha cada vez. Las primitivas **rt** y **lt** son muy importantes en LOGO, al igual que la orden **fd**. Vamos a ver cómo funciona esta última:

```
?fd 75
?rt 90
?fd 75
?
```

En la figura 2.2 puede ver que la tortuga ha avanzado una distancia de 75 unidades en sentido vertical, a continuación ha girado hacia la derecha un ángulo de 90 grados sexagesimales (que es la unidad angular que empleamos en LOGO) y finalmente ha vuelto a avanzar 75 unidades hacia la derecha en sentido horizontal. Fíjese que hemos dejado de repetir INTRO después de cada línea, pero no olvide pulsar esta tecla al final de cada una de las mismas.

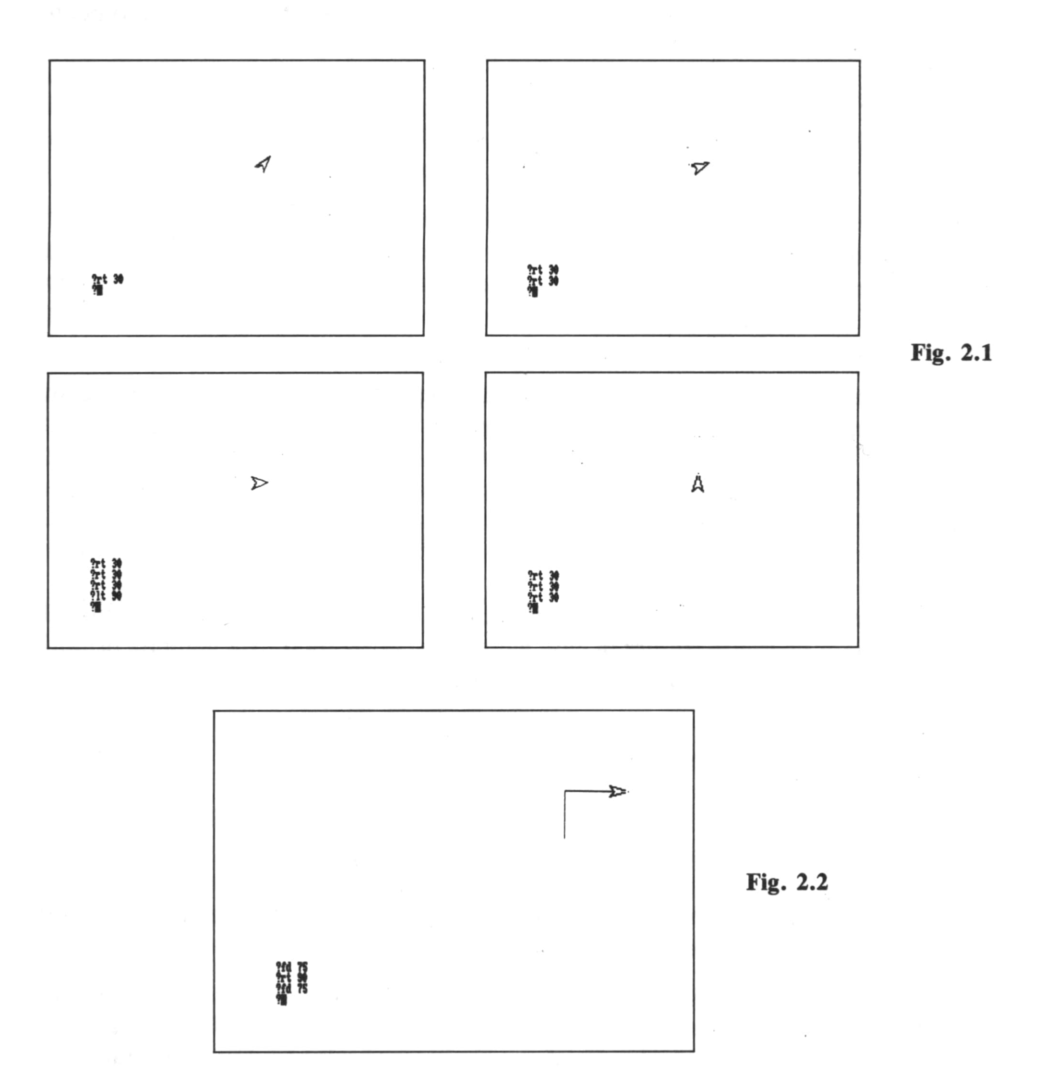


Fig. 2.1

Fig. 2.2

Continúe moviendo la tortuga con:

```
?rt 90 fd 75 rt 90 fd 75 rt 90
?
```

En este caso, la tortuga ha dibujado el perímetro de un cuadrado para quedar apuntando en el mismo sentido que cuando comenzó, es decir, hacia arriba.

El dibujo hubiera sido bastante más rápido si las primitivas **fd**, **rt** y **lt** hubiesen estado en la misma línea. Escriba:

```
?cs
?
```

que se encarga de borrar el cuadrado que había dibujado en la pantalla, y a continuación:

```
?fd 75 rt 90 fd 75 rt 90 fd 75
rt 90 fd 75 rt 90
?
```

Fíjese que, al no caber todas la primitivas en una sola línea, el ordenador salta a la siguiente y continúa escribiendo en ésta. Al final de la primera línea hay un símbolo ! para indicar que la secuencia de instrucciones no acaba ahí, sino que continúa en la siguiente línea. No he incluido este símbolo en el texto dado que, por causa del espacio libre y de otras circunstancias, puede haber pequeñas diferencias entre la presentación de las líneas en la pantalla de cada lector.

Puede continuar el dibujo con las siguientes líneas:

```
?lt 135
?fd 100 lt 90 fd 100
lt 90 fd 100 lt 90 fd 100 lt 90
?
```

La primera línea hace que la tortuga gire 135 grados a la izquierda, pero vamos a fijarnos en la segunda, que produce un resultado un tanto sorprendente.

La tortuga acaba de dibujar otro cuadrado, pero parte del mismo desaparece en el área que hasta ahora habíamos reservado para textos. Antes de hacer nada, pulse las teclas [CTRL][Y] para reproducir la línea que acaba de escribir, de forma que si ahora pulsa [INTRO] conseguirá que la tortuga vuelva sobre sus pasos y repita el dibujo del cuadrado. En todo momento, [CTRL][Y] repite la última línea que ha escrito.

Así que si ahora introduce:

```
?cs fs lt 135 [CTRL][Y]
```

conseguirá que se borre la pantalla y que aparezca un cuadrado en diagonal al igual que antes, sólo que esta vez se puede ver el cuadrado completo. Vamos a detenernos un momento para estudiar qué ha sucedido. En primer lugar hemos borrado la pantalla con **cs**; a continuación hemos pasado a una pantalla dedicada exclusivamente

a gráficos por medio de la orden **fs**. En tercer lugar hemos hecho que la tortuga gire 135 grados por medio de **lt 135**, y finalmente, con [CTRL][Y], hemos repetido las instrucciones que se encargan de dibujar el cuadrado.

La figura 2.3 muestra la versión de los cuadrados con pantalla dividida en pantalla de textos y de gráficos, mientras que la figura 2.4 muestra el cuadrado tal y como queda al final en la pantalla gráfica.

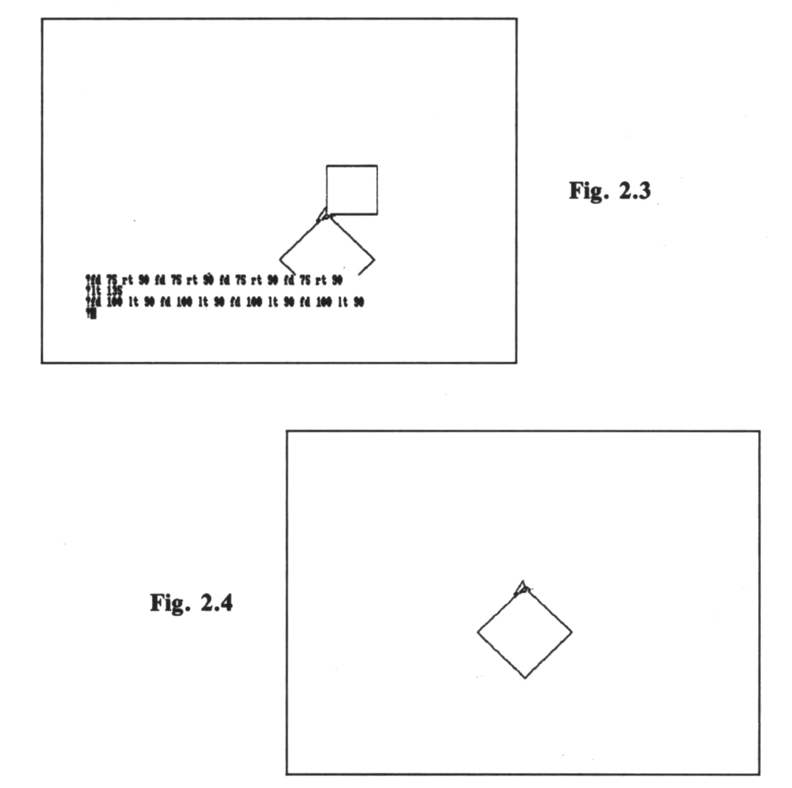


Fig. 2.3

Fig. 2.4

A continuación escriba las siguientes líneas:

```
?ss cs
?lt 45
?fd 200 rt 90 fd 200
rt 90 fd 200 rt 90 fd 200 rt 90
?
```

La primera línea devuelve a la pantalla dividida en pantalla de textos y de gráficos; la segunda línea se encarga de hacer girar la tortuga 45 grados a la izquierda. Finalmente, la tercera línea dibuja un cuadrado bastante grande, parte del cual se sale

de la pantalla, aunque la tortuga vuelve a aparecer para finalizar el dibujo. (Véase la figura 2.5)

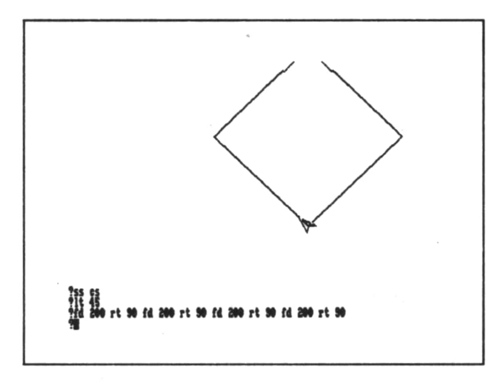


Fig. 2.5

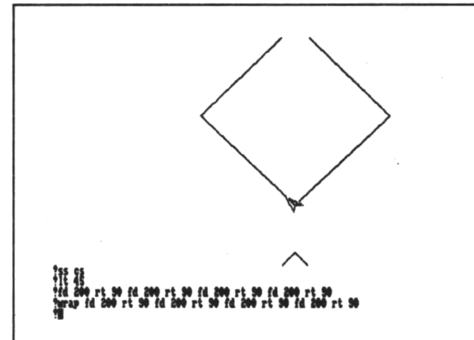


Fig. 2.6

Si ahora escribe:

?wrap CTRL Y
?

verá que todo queda como antes, pero, en esta ocasión, cuando la tortuga se sale por la parte superior de la pantalla reaparece por la parte inferior de la misma, continúa dibujando la esquina del cuadrado hasta que se vuelve a salir y, finalmente, aparece de nuevo por la parte superior. La diferencia, evidentemente, procede de la orden **wrap**. Esta primitiva hace que la tortuga reaparezca por el lado de la pantalla opuesto a aquél por el que sale, ya sea por arriba, por abajo o por los laterales. La figura 2.6 nos muestra el resultado.

Haga otro experimento mientras continúa bajo los efectos de **wrap**:

?cs
?rt 25 fd 999
?

con lo que se borra la pantalla, la tortuga gira y traza una recta.

Sólo dibuja una recta, es cierto, pero fíjese que después de desaparecer por arriba y de reaparecer por abajo continúa saliendo y entrando en la pantalla. Pruebe varias distancias y ángulos de giro; seguro que consigue efectos muy interesantes.

Ya he mencionado que este efecto se denomina "wrap around", que en inglés significa "envolver". Puede volver al modo de dibujo normal por medio de la orden **window**. Vamos a ver las diferencias entre ambos sistemas de dibujo. Escriba:

```
?window
?ss cs
?rt 25 fd 999
?
```

En esta ocasión la tortuga gira, dibuja una recta hasta salirse de la pantalla y ya no vuelve. Esta primitiva se denomina **window** porque considera la pantalla gráfica como una ventana que nos permite ver parte de una zona de dibujo mucho mayor. Normalmente, al trabajar en Dr LOGO se emplea este modo de dibujo.

Existe otra forma de controlar los movimientos de la tortuga; para ello puede emplearse la primitiva **fence**, que en inglés significa cerca o valla. Esta primitiva impide que la tortuga se salga de los bordes de la pantalla gráfica. Pruebe el siguiente ejemplo:

```
?cs fence rt 25 fd 999
Turtle out of bounds
?
```

Esta vez la tortuga no se ha movido de su sitio. **fence** hace que la tortuga calcule previamente a dónde va a ir a parar al realizar las instrucciones de una primitiva. En caso de que tenga que salirse de la pantalla, ni siquiera comienza el movimiento.

En todo el capítulo siguiente vamos a trabajar en modo **wrap**, así que, para disponer de una hoja limpia para nuestros próximos experimentos, escriba:

```
?bye
```

con lo cual se encontrará de nuevo con:

```
A>
```

Para volver a LOGO desde CP/M no tiene más que escribir:

```
A>logo
```

y ya está en disposición de pasar al próximo capítulo.

Capítulo 3

Procedimientos en LOGO

Hasta ahora ha tenido que escribir un montón de instrucciones para conseguir unos efectos que realmente no son muy espectaculares que digamos: un par de cuadrados nada más. Además, si su habilidad como mecanógrafo es parecida a la mía, seguramente estará empezando a pensar que va a estar toda la vida cometiendo errores, pero no se preocupe.

Lo más importante de LOGO no es precisamente los gráficos de tortuga; además los que hemos visto hasta ahora quedan muy por debajo de las posibilidades de LOGO. No cabe duda de que los gráficos de tortuga son importantes, pero no hay que sobrevalorarlos. La capacidad de proceso de listas tampoco es lo más importante de LOGO; posiblemente usted ni siquiera haya oído hablar del tema. La característica más importante de LOGO es su capacidad de emplear procedimientos que son equivalentes a las primitivas, con la única diferencia de que los procedimientos los define el usuario.

En primer lugar, con ellos se consigue reducir bastante la cantidad de instrucciones que deben escribirse. Aunque es un objetivo secundario, reconocerá que adquiere bastante importancia si se fija, por ejemplo, en la cantidad de trabajo ahorrado al usar una sola palabra, como puede ser **fd**.

3.1 Tortugas viajeras

En el capítulo anterior, ha tenido que escribir cuatro veces las instrucciones **fd 100 rt 90** para dibujar un cuadrado. La instrucción completa fue tan larga que el ordenador tuvo que saltar a otra línea de pantalla para completar la sentencia, aunque finalmente trató ambas líneas como si fuesen una sola. Cualquier lenguaje con un poco de lógica debe tener en cuenta el carácter repetitivo de las instrucciones, en nuestro caso **fd 100 rt 90**. Dr LOGO es un lenguaje bastante lógico.

En LOGO no tiene más que escribir el par de primitivas **fd 100 rt 90** una sola vez y solicitar a continuación al ordenador que las repita cuatro veces. Escriba:

```
?cs
?repeat 4 [fd 100 rt 90]
?
```

Al introducir la primera línea, **cs**, la pantalla se divide en pantalla de textos y de gráficos (a no ser que previamente hubiese solicitado que la pantalla completa se dedicase a pantalla gráfica). Al introducir la segunda línea es cuando el ordenador dibuja el cuadrado.

El ordenador ha repetido cuatro veces todo lo que se encuentra dentro de los corchetes, a lo que suele denominarse "lista". Es costumbre encerrar entre corchetes todos los elementos de la lista, en nuestro caso **[fd 100 rt 90]**. En esta ocasión puede decirse que se trata de una lista de instrucciones.

A continuación vamos a ver un ejemplo similar que cuenta con una lista de instrucciones prácticamente idéntica a la anterior:

```
?cs repeat 360 [fd 1 rt 1]
?
```

Al observar el lento avance de la tortuga no puedo evitar que me venga a la memoria la fábula de Esopo "La Liebre y la Tortuga"; creo que la de Esopo correría más que la nuestra. Esta vez la tortuga recorre un camino circular, pero aunque el dibujo final parece una circunferencia, realmente no lo es. No es más que un polígono regular de 360 lados.

No debe sorprender que el dibujo final se parezca tanto a una circunferencia; vamos a dar un par de vueltas a un viejo tópico. Un polígono regular de nueve lados se parece más a una circunferencia que, por ejemplo, un cuadrado, así que un polígono de 360 lados debe parecerse mucho más todavía. Como en muchas otras ocasiones, es muy difícil distinguir la realidad de algo que no es más que una aproximación a la misma.

Antes de borrar la circunferencia que tiene en pantalla, dé las siguientes instrucciones:

```
?repeat 360 [fd -1 rt 1]
?
```

Esta vez la tortuga dibuja otra circunferencia a la izquierda de la anterior, con lo que obtendrá una imagen que recuerda a un buho que esté mirando. La figura 3.1 muestra el buho... un poco vizco, ¿verdad?

Si quiere que la tortuga sea un poco más rápida, escriba

```
?cs
?ht
?repeat 360 [fd 18 rt 18]
?
```

y sin esperar siquiera a despedirse, la tortuga desaparece como consecuencia de la orden **ht**, que significa "hideturtle" (ocultar la tortuga). De nuevo obtiene la cir-

cunferencia anterior, esta vez un poco más rápido, aunque de todas formas la velocidad no ha sido ninguna maravilla. Ya sabe que esta circunferencia realmente no es más que un polígono regular de 360 lados, pero ¿realmente son necesarios tantos lados para obtener una circunferencia aceptable?

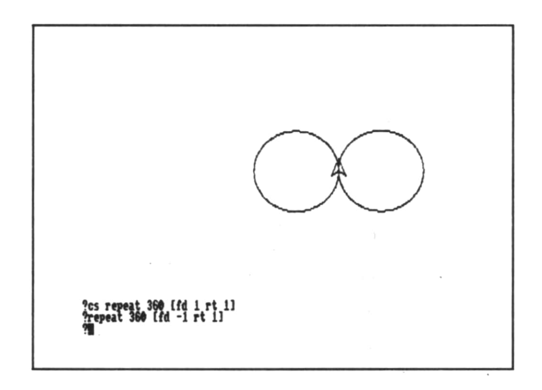


Fig. 3.1

La figura 3.2 muestra un polígono de 20 lados que también se parece bastante a una circunferencia. Para obtener una "circunferencia" de 20 lados, aproximadamente del mismo tamaño que el de 360 lados, hay que modificar la distancia de avance, es decir, el parámetro de la primitiva **fd**, y el ángulo de giro. La nueva circunferencia tiene 20 lados, esto es, 18 veces menos que el de 360, así que cada lado debe ser unas 18 veces más largo que los lados del original. Por tanto, se debe realizar 20 veces el par de instrucciones **fd 18 rt 18**.

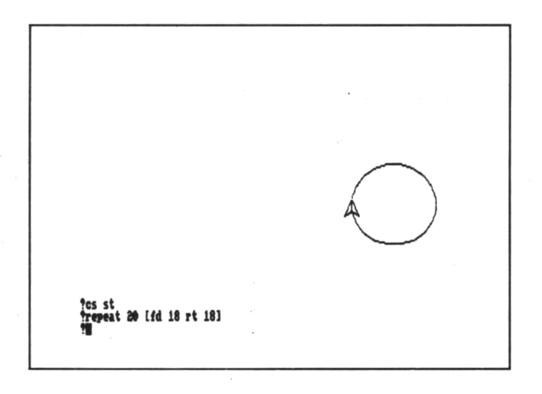


Fig. 3.2

Para obtener la circunferencia/polígono de 20 lados escriba:

```
?cs st
?repeat 20 [fd 18 rt 18]
?
```

El resultado es la figura 3.2. El ordenador ha dibujado esta circunferencia en mucho menos tiempo que en el primer ejemplo, dado que ha tenido que realizar bastantes menos cálculos: unas veinte veces menos. Fíjese que la primitiva **st**, que quiere decir "showturtle" (mostrar la tortuga), ha vuelto a colocar la tortuga en la pantalla.

Si deja en pantalla la "circunferencia" de 20 lados y escribe las siguientes líneas:

```
?ht
?repeat 20 [bk 18 rt 18]
?
```

el ordenador dibuja otro círculo a la izquierda del anterior. Esta vez no habrá visto a la tortuga recorrer su camino; probablemente lo haya hecho, pero al quitar la tortuga de pantalla con **ht**, no se ha dado cuenta. La orden **bk 18** (retroceder 18 unidades) es equivalente a **fd 18**, pero en sentido contrario. La figura 3.3 muestra las dos circunferencias de 20 lados; es interesante compararlas con las de 360 lados.

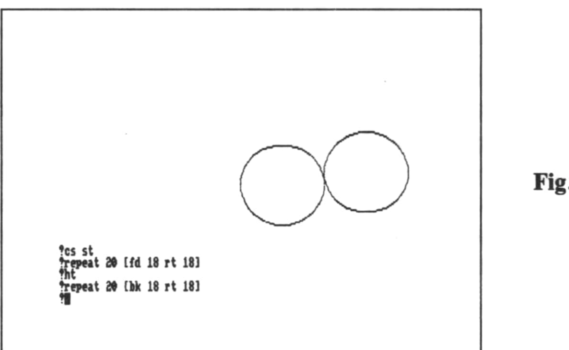


Fig. 3.3

Intente practicar con este sistema de repetición de instrucciones para dibujar un polígono de 10 lados y otro de 5 lados. En seguida comprobará si lo ha conseguido, porque en ese caso la respuesta debe aparecer en pantalla. De todas formas, antes de intentarlo escriba lo siguiente:

```
?st
```

para hacer que la tortuga vuelva a la pantalla.

3.2 Estrellas fugaces

Hasta ahora hemos conseguido dibujar varias circunferencias por medio de instrucciones del tipo **fd :longitud rt :angulo**, en las que las variables **:longitud** y **:angulo** cambiaban en cada ejemplo. Vamos a emplear el mismo sistema con las siguientes instrucciones:

```
?cs
?st
?repeat 5 [fd 100 rt 144]
?
```

Esta vez el resultado no ha sido una circunferencia, sino una estrella. Se trata de un pentágono estrellado, y no se parece en absoluto a un círculo. (Véase la figura 3.4)

Vamos a probar las siguientes primitivas para dibujar una estrella de ocho puntas:

```
?cs
?repeat 8 [fd 100 rt 135]
?
```

Antes de seguir adelante vamos a intentar averiguar de dónde ha salido todo esto. Es conveniente que compare su estrella con la que aparece en la figura 3.5.

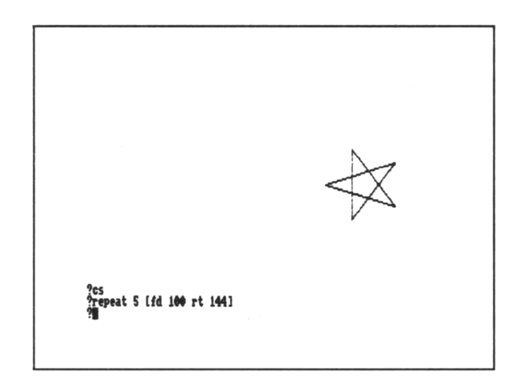


Fig. 3.4

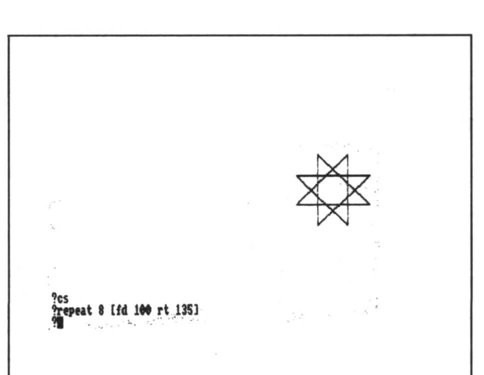


Fig. 3.5

En ambas estrellas se ha repetido una lista de instrucciones, pero con una diferencia fundamental: el ángulo girado.

También es diferente el número de veces que se repite la lista de instrucciones, pero esto no tiene tanta importancia. Además es conveniente fijarse en un pequeño detalle: si calcula 5*144 obtendrá 720; por su parte 8*135 son 1440. Tanto 720 como 1440 son múltiplos de 360. ¿A qué se debe esta coincidencia y por qué ambos valores son múltiplos diferentes de 360?

Ahora escriba lo siguiente:

```
?cs repeat 200 [fd 100 rt 135]
?
```

y observe que la tortuga vuelve a dibujar la estrella de ocho puntas, pero esta vez se pasea un buen rato por encima de las líneas que forman la estrella antes de que reaparezca el signo de interrogación. Cuando, finalmente, la tortuga se queda quieta y la interrogación vuelve a aparecer en la parte baja de la pantalla que reservamos para textos, fíjese que el dibujo obtenido es exactamente igual al conseguido únicamente con 8 repeticiones. Esta vez la tortuga ha recorrido el dibujo 25 veces, dado que 200/8=25. Sin embargo, una vez que la tortuga recorre por primera vez la estrella, las repeticiones no afectan para nada a la figura.

Si ahora modifica la línea anterior (el sistema más comodo es emplear [CTRL][Y], las teclas de movimiento del cursor y la de borrado) para dejarla así:

```
?cs ht repeat 200 [fd 100 rt 135]
?
```

volverá a obtener la misma estrella. Parece que no ha cambiado nada, hasta que vuelve a aparecer la interrogación. Puede volver a ejecutar la misma línea y detener la repetición del dibujo en cualquier momento pulsando simultáneamente las teclas **CTRL** y **G** (en el futuro lo indicaremos abreviadamente con [CTRL][G]). El ordenador deja de dibujar y LOGO envía el siguiente mensaje:

```
Stopped!
?
```

El mismo efecto se consigue pulsando la tecla [ESC]. Si modifica un poco la línea anterior:

```
?cs repeat 200 [fd 150 rt 75]
?
```

esta vez habrá obtenido una figura bastante más complicada (véase la figura 3.6), que no hace más que confirmar que, lo que en un principio puede parecer bastante

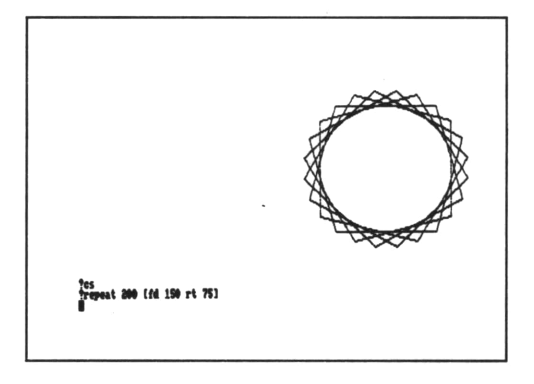


Fig. 3.6

complicado de programar, en la práctica puede ser de lo más sencillo. La figura que ha obtenido no es más que una estrella con muchas puntas, para lo cual se ha repetido muchas veces la secuencia de instrucciones que hace que la tortuga avance y gire a la derecha.

3.3 Cómo definir una estrella

Aunque es posible estudiar las consecuencias de emplear diferentes ángulos y longitudes sin más que escribir variantes de las instrucciones anteriores, hay que tener en cuenta que LOGO ha sido creado de forma que haga las cosas lo menos complicadas posible; con este lenguaje se obtiene el mismo resultado con un método mucho más sencillo.

En primer lugar, escriba la siguiente línea; fíjese cómo cambia el mensaje inductor, es decir, la interrogación:

```
?to estrella :lado :angulo
>
```

Es este momento aparece otro mensaje inductor, el símbolo >, que le indica que se encuentra en modo de definición. LOGO pasa a modo de definición al recibir la orden **to**.

Ahora escriba el siguiente texto, teniendo cuidado de no equivocarse. Para dejar las cosas un poco más claras he repetido la primera línea con el símbolo "?".

```
?to estrella :lado :angulo
>repeat 200 [fd :lado rt :angulo]
>end
estrella defined
?
```

La parte comprendida entre **to** y **end** es la definición del procedimiento llamado **estrella**. LOGO reconoce el final de dicho procedimiento al encontrarse en una línea únicamente la palabra **end**. Por su parte, sabrá que todo ha funcionado correctamente cuando LOGO envíe el mensaje **estrella defined**; a continuación LOGO hace que vuelva a la pantalla el mensaje inductor habitual, la interrogación, haciendo desaparecer el símbolo ">".

Finalmente, para comprobar que ha realizado la definición correctamente no tiene más que escribir:

```
?po "estrella
to estrella :lado :angulo
repeat 200 [fd :lado rt :angulo]
end
?
```

aunque si comete un error, por mínimo que sea, como por ejemplo:

?po estrella
Not enough inputs to estrella in estrella: po estrella
?

LOGO le dirá que algo no marcha bien.

De momento es suficiente con que tenga en cuenta que:

1. Un símbolo de dos puntos (:) delante de un nombre indica que estamos hablando del valor de la variable de que se trate.
2. Unas comillas (") delante de un nombre indican que nos referimos al nombre en sí.
3. El nombre solo, sin que le acompañe ningún símbolo, indica que nos referimos a la acción de un procedimiento.

Al pedir al ordenador un listado de la definición de **"estrella** por medio de la orden **po**, LOGO interpreta que estamos hablando de algo que se llama **estrella**.

Si cometemos un error y solicitamos al ordenador un listado de **estrella** sin incluir las comillas, LOGO piensa que nos estamos refiriendo a la acción del propio procedimiento llamado **estrella**. Esto explica su extraña conducta al decirnos que faltan datos por medio del mensaje "**Not enough inputs...**", dado que LOGO supone que lo que le estamos pidiendo es que realice el procedimiento **estrella**, y una parte del procedimiento en sí es tomar los datos necesarios. Más adelante seguiremos hablando de este tema.

En caso de que haya cometido algún error al definir **estrella**, deberá editar el texto para corregirlo. Tendrá que aprender antes o después cómo se edita un procedimiento en LOGO, así que, aunque no haya cometido ningún error, escriba la siguiente línea:

?ed "estrella

con lo que la pantalla queda vacía y se entra en el modo de edición (véase la figura 3.7). En la parte superior de la pantalla verá:

to estrella :lado :angulo
repeat 200 [fd :lado rt :angulo]
end

Ahora se puede modificar la definición de **estrella** empleando para ello las teclas de movimiento del cursor, las de borrado y los caracteres de inserción. Al finalizar las modificaciones pulse COPY para salir del modo de edición, con lo que recibirá el mensaje:

estrella defined
?

Conviene que practique cómo se pasa del modo normal al de edición y a la inversa; si no quiere volver a copiar el procedimiento con COPY puede emplear la tecla ESC, con lo que obtendrá:

Stopped!
?

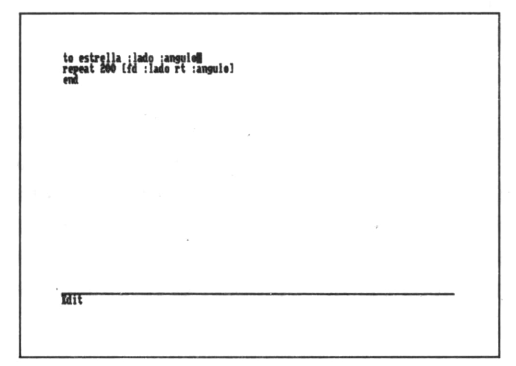


Fig. 3.7

3.4 Ha nacido una estrella

Hay dos valores que van ligados al procedimiento **estrella**: el **:lado** y el **:angulo**. Con el primero se fija la longitud que debe avanzar la tortuga, mientras que el segundo define su ángulo de giro.

A continuación vamos a ver de nuevo el procedimiento **estrella** para que sirva de referencia:

```
to estrella :lado :angulo
repeat 200 [fd :lado rt :angulo]
end
```

Junto a la instrucción **fd**, que ya conocemos, nos encontramos la variable **:lado**, así que **:lado** debe representar una longitud. Por su parte, **:ángulo** aparece junto con la instrucción **rt**, así que debe tratarse de un ángulo. Juegue un poco con **estrella**; dé las siguientes instrucciones:

```
?cs
?estrella 100 144
?
```

con lo que debe obtener la estrella de cinco puntas que ya le es familiar, porque ha movido la tortuga con la instrucción **fd 100 rt 144** al hacer **:lado** igual 100 y **:angulo** igual a 144. El procedimiento **estrella** necesita dos datos que le son enviados en

dos variables, **:lado** y **:angulo**. LOGO comienza a ejecutar el procedimiento y sustituye **:lado** por su valor en todos los lugares en que aparezca. Los datos que aparecen en un procedimiento también se denominan "parámetros".

En el resto del capítulo vamos a emplear la orden **wrap** que hemos visto anteriormente; para ello escriba las siguientes líneas:

```
?wrap cs
?estrella 100 144
?estrella 150 144
?
```

que mantienen a la tortuga ocupada durante un buen rato dibujando dos estrellas de cinco puntas, una mayor que la otra, dado que la primera tiene lados de 100 unidades de longitud mientras que la segunda tiene lados de 150 unidades. Fíjese que ambas formas son iguales, únicamente se diferencian en el tamaño; esto se debe a que el primer parámetro (**:lado**) pasa de valer 100 a 150 mientras que el segundo (**:ángulo**) vale 144 en ambos casos. Compare el resultado que ha obtenido con la figura 3.8.

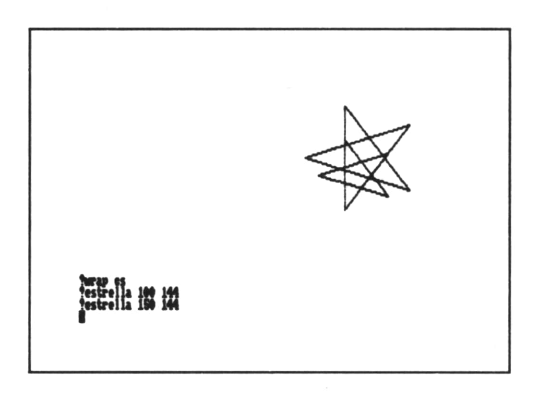


Fig. 3.8

Es muy probable que usted no haya conseguido definir el procedimiento sin cometer ningún error al escribir las instrucciones. A veces puede darse cuenta de los errores, pero otras veces no...

```
?cs
?estrella 100 144
I don't know to repear in estrella: repear 200
[fd :lado rt :angulo]
?
```

Aunque no lo parezca, LOGO envía estos mensajes con la intención de colaborar en lo que pueda. Con este mensaje LOGO indica que no sabe qué quiere decir la

palabra **repear**, pero ¿dónde está y qué significa **repear**? Siempre que reciba el mensaje **I don't know how to...**, que quiere decir "No sé como hacer ...", conviene escribir inmediatamente:

```
?ed
```

con lo que se encontrará de nuevo en el editor. LOGO indica con el cursor dónde se encuentra el posible error. En nuestro ejemplo la pantalla sería la siguiente:

```
to estrella :lado :angulo
repear 200 [fd :lado rt :angulo]
end
```

con lo cual nos damos cuenta de que hemos introducido **repear** en lugar de **repeat** y se ha producido el error; LOGO no sabe que singifica **repear**. Modificar la letra en cuestión es una tarea sencillísima, así que, ¿por qué no hace unos cuantos experimentos con el editor creando errores parecidos?

Otro error posible es el siguiente:

```
?cs
?estrella 100 144
I don't know how to rt:angulo in estrella: repeat
200 [fd :lado rt:angulo
?
```

que hace que la tortuga se detenga nada más dibujar la primera línea ascendente. Hemos omitido un elemento muy importante: el espacio entre **rt** y **:ángulo**. En LOGO colocamos un espacio entre las distintas palabras para separarlas.

Dependiendo de las circunstancias en que nos encontremos, hay otras formas de separar las palabras que veremos un poco más adelante al hablar de los "delimitadores". Una de las formas de separación con el editor es emplear corchetes cuadrados, []; esta característica tiene en cuenta que en cualquier momento podemos necesitar editar varios procedimientos, cuyos nombres colocaremos en la lista. Los corchetes cuadrados delimitan una lista.

```
?ed[estrella]
```

La línea anterior también podía haber sido escrita así:

```
?ed [estrella]
```

con un espacio entre la primitiva y el corchete, con lo que queda bastante más fácil de leer.

En el caso de **I don't know how to rt:angulo in estrella:...** el editor presentará la siguiente pantalla:

```
to estrella :lado :angulo
repeat 200 [fd :lado rt:angulo]
end
```

que no debe sorprenderle lo más mínimo si ha seguido hasta aquí todo el proceso.

Ahora escriba la siguiente línea:

```
?estrella100 144
```

Independientemente de que haya definido el procedimiento correctamente, habrá obtenido un error, que en este caso en particular es:

```
I dont't know how to estrella100
?
```

que le indica que falta un espacio entre **estrella** y **100.** La mayor parte de los errores que se suelen producir en LOGO no son más que la omisión de espacios entre los componentes de una sentencia.

3.5 Datos y parámetros

Hay otro error bastante frecuente que también se debe a la omisión de un espacio entre dos elementos; es el siguiente:

```
?estrella 100144
Not enough inputs to estrella in estrella: estrella 100144
?
```

Este error se debe a que Dr LOGO esperaba varios números después de **estrella**; el procedimiento emplea dos parámetros y en este caso no hemos enviado más que un dato, 100144, al olvidar el espacio que separa los dos valores. Recordemos que "parámetro" es el nombre técnico de cada uno de los datos con que trabaja el procedimiento.

El procedimiento que hemos denominado **estrella** emplea dos parámetros que deben incluirse a continuación del nombre del procedimiento; se trata de las variables que hemos llamado **:lado** y **:angulo**. De la misma forma, la primitiva **fd** tiene un solo parámetro: el número que sigue a la instrucción y que representa la longitud de avance. Por ejemplo, en **fd 90** el valor del parámetro es 90 unidades. Las primitivas **rt** y **lt** emplean también un solo parámetro, que en ambos casos representa el ángulo de giro.

La primitiva **repeat**, que también forma parte de la definición del procedimiento, es bastante más compleja que **fd** o **rt**. Esta primitiva necesita dos parámetros. Por ejemplo, en **repeat 360 [fd 1 rt 1]** el primer parámetro es 360 e indica el número

de veces que queremos que se repita la lista de instrucciones que aparece a continuación. El segundo parámetro es la propia lista de instrucciones, y es un poco más complicado que una letra o un número.

Cuando ejecutamos el procedimiento, como por ejemplo en el siguiente caso:

```
?estrella 100 144
?
```

LOGO sustituye realmente el nombre del parámetro **:lado** por el valor 100 en todo el procedimiento. Así, al leer **fd :lado** LOGO interpreta **fd 100**. Como el parámetro **:angulo** toma el valor 144, **rt :angulo** es equivalente a **rt 144**.

El valor 100 que hemos asignado a **:lado** al ejecutar el procedimiento no es válido más que para dicho procedimiento, es decir, si tenemos otra sentencia que contenga el término **:lado** fuera de **estrella**, su valor no resulta afectado. Por esta causa se dice que los parámetros tienen validez "local" o que están limitados al procedimiento en cuestión.

Dada la flexibilidad que permite el uso de parámetros, podemos probar cualquier combinación de números con el procedimiento **estrella**. Vamos a probar el siguiente par de parámetros:

```
?wrap fs cs estrella 800 175
?
```

Gracias a los efectos que crea la primitiva **wrap**, se obtiene un dibujo bastante curioso que ilustramos en la figura 3.9. En una ocasión mi hija y una amiga suya inventaron el "Juego de los Teléfonos" haciendo experimentos con el procedimiento **estrella**.

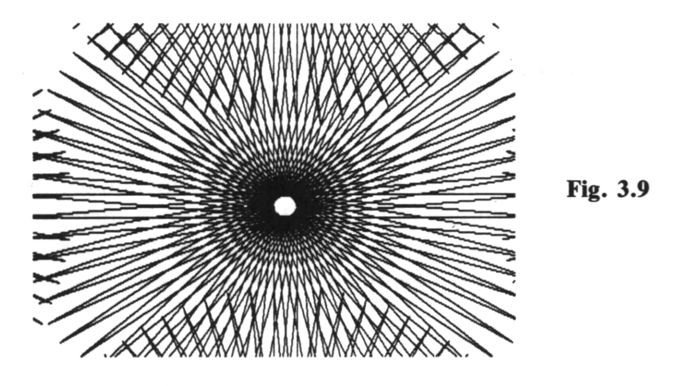


Fig. 3.9

El juego consiste en tomar los últimos siete dígitos de un número de teléfono y colocarlos como parámetros al ejecutar el procedimiento **estrella**; hay quien dice que

el resultado tiene una cierta semejanza con el carácter del propietario del número en cuestión. Por ejemplo, el número de Amsoft en Inglaterra es 027 230222; tomamos los últimos siete dígitos y los colocamos en la siguiente sentencia:

```
?fs cs estrella 723 0222
?
```

El número de teléfono de Amsoft demuestra que su aspecto es tan brillante como el de la figura 3.10.

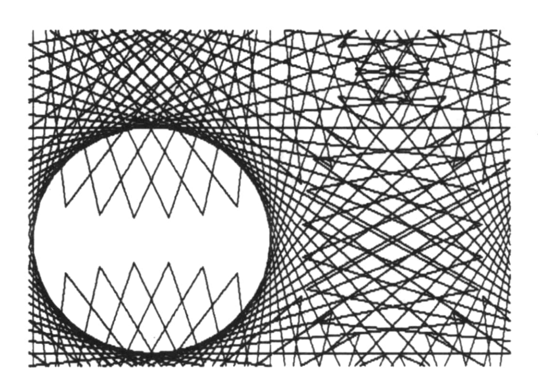


Fig. 3.10

Juegue con otros números de teléfono para ver si el resultado le recuerda a alguien en particular. Es curioso cómo algunas personas dan la misma sensación de barullo que sus números de teléfono.

Ahora que ya tiene un poco de experiencia en lo fácil que resulta cometer errores, puede incluir un tercer parámetro en el procedimiento **estrella**: el número de veces que debe repetirse la lista de instrucciones. En esta ocasión la primera línea de la definición pasa a ser la siguiente:

```
?to estrella :lado :angulo :veces
>
```

No se preocupe en caso de que no le salga bien; en el próximo capítulo volveremos a tratar en profundidad este procedimiento deteniéndonos a estudiar qué sucede cuando cometemos errores en LOGO. Pero, antes de adentrarnos en este tema, vamos a desviarnos ligeramente de nuestro camino para estudiar cómo grabar en disco nuestros pequeños logros. También vamos a ver qué relación hay entre LOGO y CP/M.

3.6 Cómo grabar en disco el trabajo realizado

Hasta ahora ha escrito unos cuantos procedimientos bastante cortos que el ordenador guarda en la zona de memoria reservada para procedimientos y variables. En

esta misma zona el ordenador guarda el proceso seguido para crear sus programas.

Puede averiguar cuánta memoria queda libre sin más que escribir la palabra **nodes**:

```
?nodes
1578
?nodes
1575
?
```

Al preguntar al ordenador por primera vez cuánta memoria queda, la respuesta es 1578 nodos (seguramente el valor que usted ha obtenido no es el mismo); no confunda el número de nodos con la memoria en bytes; realmente es mucho más sencillo medir la cantidad de memoria en nodos. Al realizar la segunda pregunta, nos damos cuenta de que el número de nodos ha descendido tres unidades simplemente por haber formulado la pregunta ¿Esto quiere decir que nos vamos a quedar sin memoria según vamos dando instrucciones al ordenador?

Ejecute la siguiente instrucción:

```
?repeat 2000 [pr nodes]
```

En la pantalla aparece una sucesión de números que van descendiendo lentamente hasta llegar aproximadamente a cien nodos. En a este punto LOGO se detiene, parece que se queda pensando un rato y a continuación sigue con la serie de números descendentes, sólo que ha vuelto a comenzar desde un número bastante más alto. Cuando LOGO se detuvo, lo que realmente estaba haciendo era buscar todas las posiciones de memoria reutilizables.

LOGO cuenta con una primitiva con la que puede solicitarse una reorganización de la memoria:

```
?nodes
1123
?recycle nodes
1730
?
```

Cuando LOGO recibe la orden **recycle**, comienza a recuperar posiciones de memoria y a reorganizar la misma, para lo cual se produce una pequeña pausa. La próxima vez que pregunte cuántos nodos quedan libres obtendrá una respuesta bastante más alta que antes.

Al conectar el ordenador y comenzar a trabajar con LOGO, el número de nodos es superior a dos mil; pero, a medida que van creándose procedimientos y asignándose variables, el número de nodos disminuye y no volverá al valor inicial aunque se utilice la instrucción **recycle**. LOGO consume nodos para almacenar la información que va introduciendo; hay que tener en cuenta que cuando da a LOGO de Ams-

trad la orden de grabar la información en disco, éste graba toda la información indiscriminadamente. Esto quiere decir que para grabar sus procedimientos tendrá que grabar todo lo que hay en memoria.

Grabe sus primeros procedimientos llamando **ejemplo** al fichero en que va a colocarlos. Escriba la siguiente secuencia de instrucciones:

```
?save "ejemplo
?dir
[EJEMPLO]
?load "ejemplo
xxx defined
yyy defined
?
```

dir es una primitiva de LOGO que nos da una lista de los ficheros LOGO que se encuentran en el disco, mientras que **load** es la orden que nos permite leer un fichero de procedimientos. La línea **xxx defined** nos informa de los procedimientos al leer sus definiciones en el disco.

Es conveniente que lea el apéndice ''Discos y LOGO'', en el que aparecen más detalles sobre este mismo tema.

Capítulo 4

Exploración de LOGO

Hay un viejo dicho que con frecuencia ignoramos u olvidamos, y la verdad es que es absolutamente cierto: ''Aprendemos de nuestros errores''.

Además, con mucha frecuencia pensamos que un error es algo malo, y en caso de cometerlo casi siempre intentamos ocultarlo o ignorarlo; casi nunca estudiamos sus causas y consecuencias. Hay otro dicho popular, ''Errar es humano, olvidar divino'', pero esto no quiere decir que tengamos que echar tierra sobre nuestros errores como si fuesen algo condenable.

Los diseñadores de LOGO intentaron conseguir un lenguaje que ayudara al usuario a aprender de los errores que cometiese al explorar las características del lenguaje, y lo han conseguido.

4.1 Aprender de los errores

Los creadores de LOGO se dieron cuenta desde un principio de que, para sacar algo en claro de nuestras propias equivocaciones, es preciso que la información que nos dé LOGO sobre las mismas sea lo más constructiva posible. En algunas ocasiones puede parecer que la ayuda que nos proporciona LOGO no es ni mucho menos tan útil como cabría esperar, pero no hay duda de que la intención de LOGO es en todo momento ayudar en lo que pueda. Recuerde que los creadores de LOGO también son humanos.

LOGO da mucha importancia a este tipo de ayuda constructiva. Todos confiamos en que con el paso del tiempo cometeremos menos errores, pero también es de esperar que cada vez nos ocupemos de cosas más difíciles que exijan más atención. Aunque no cabe duda de que con el tiempo irá avanzando en el arte de la programación, sus avances no servirían de nada si al cometer una equivocación no cuenta con nada que le explique cuál es el origen de la misma. No es que LOGO le anime a cometer errores, lo que realmente intenta es inducirle a emprender tareas más complicadas, echándole una mano cada vez que se equivoque... y sin duda siempre surgirá algún fallo.

LOGO tiene como objetivo desarrollar sus capacidades como programador, ayudándole al mismo tiempo a realizar las tareas que le apetezca; por tanto, el hecho

de tener que acudir constantemente a un programador con más experiencia para que le ayude a descubrir los gazapos que ha cometido, va en contra de este objetivo. Los diseñadores de LOGO creen firmemente que aprendemos fundamentalmente a partir de nuestro propio esfuerzo. Por esta misma razón no busque en esta Guía respuestas definitivas y concluyentes; por el contrario, tenemos la intención de que le anime a plantearse preguntas interesantes y a responderlas por sí mismo.

Desde luego, no se trata de prescindir totalmente de las respuestas y sugerencias que puede proporcionar un lenguaje de programación. Los creadores de LOGO han pretendido que usted haga pruebas y experimentos y que cuente con la posibilidad de cometer errores para aprender de ellos, introduciéndose en un proceso constructivo, que es la depuración de sus propios programas.

Uno de los creadores de LOGO, Seymour Papert, ha señalado con toda razón: "... todos aprendemos de nuestros propios errores, pero en LOGO la pregunta más importante no es "¿Funciona correctamente?", sino "¿Cómo puedo arreglarlo?", aunque, por supuesto, es muy importante que al final el programa funcione". (De Mindstorms, Harvester Press).

Trate de convencerse de que cometer errores no es tan importante como disfrutar corrigiéndolos; este proceso es un ejercicio formativo muy importante. Aprender a corregir errores propios es una forma de aprender a resolver problemas, y los expertos en la resolución de problemas son muy apreciados en nuestra sociedad actual. Es cierto que estos expertos son muy escasos hoy en día; probablemente esto se deba a que desde pequeños nos han enseñado a tener miedo a los errores y a sentirnos culpables cuando los cometemos. Un miedo exagerado a los propios fallos ahoga la innovación y la creatividad.

4.2 Cómo descubrir nuestros errores

Cada vez que LOGO no comprende sus instrucciones lo comunica por medio de un mensaje.

Cuando esté usando LOGO puede suceder que diga algo al ordenador que, a pesar de todas las palabras que para él tienen significado, sea una bobada. Escriba la palabra **bobada** nada más poner en marcha LOGO para ver qué sucede:

```
?bobada
I don't know how to bobada
?.contents
```

A continuación aparece en pantalla una lista bastante larga con los contenidos del área de memoria reservada para LOGO; en algún lugar de esta lista debe aparecer la palabra **bobada**. A pesar de que LOGO no reconozca esta palabra como una instrucción, primitiva, procedimiento o variable, el ordenador la ha incorporado al área de trabajo.

Lo que ha sucedido es que LOGO, o la parte del sistema que denominamos "intérprete", sabe que ha pulsado una serie de caracteres que constituyen la palabra **bobada**. El intérprete de LOGO tiene en todo momento una lista de las palabras que reconoce el lenguaje, en la que junto a cada palabra aparece una descripción de lo que se supone que debe suceder cuando la empleamos. Esto quiere decir que cada palabra cuenta con ciertas propiedades; en capítulos posteriores veremos más detalles sobre este tema. Por ahora la palabra **bobada** no tiene ninguna propiedad. Las palabras reconocidas, junto con sus propiedades, constituyen el "área de trabajo" de LOGO.

Desde el momento en que entra en funcionamiento, LOGO conoce las propiedades de unas palabras determinadas; cada vez que le decimos una "bobada" al ordenador, LOGO la toma como una palabra nueva y la guarda en su lista.

Cuando se introduce la palabra **bobada**, el intérprete de LOGO no la encuentra en su lista, pero a partir del momento en que se ha empleado la palabra por primera vez, incluso de una forma accidental, LOGO añade **bobada** a su lista (aunque sin asignarle ninguna propiedad). Como es lógico, LOGO no puede emplear la palabra **bobada** porque no se le ha dicho cuáles son las propiedades asociadas a dicha palabra.

Por eso introducir una palabra, en este caso **bobada**, sin decir antes a LOGO qué es lo que tiene que hacer con ella, constituye un error de programación. A pesar de todo, LOGO intenta explicar por qué no puede colaborar con nosotros. Se trata de un lenguaje muy benévolo, ¿verdad?

Ahora escriba la siguiente línea:

```
?fd 60 fd60
```

La pantalla se borra, LOGO dibuja una línea y le envía un mensaje. Responda con otra línea:

```
I don't know how to fd60
?ts .contents
```

En la respuesta hemos incluido **ts** (pantalla de textos) para que los contenidos de la pantalla no se salgan por la parte inferior de la misma, como sucedería si trabajásemos en una pantalla dividida. Si realiza un examen con detenimiento se dará cuenta de que el intérprete ha añadido la palabra **fd60** a la lista.

Después de realizar una sesión de trabajo larga con LOGO, es interesante analizar los contenidos del área de trabajo, con **.contents** al igual que antes, para ver cuántos errores de mecanografía se ha cometido. La diferencia entre **fd 60** y **fd60** es evidentemente el espacio que separa **fd** y **60**. El uso de espacios en LOGO tiene una importancia crucial; no se puede colocar espacios dentro de una misma palabra, por eso hay que emplear el subrayado, como por ejemplo "**dos_palabras**" y no "**dospalabras**".

En el ejemplo anterior, LOGO reconoce **fd** como una primitiva cuya función es "mover la tortuga hacia delante el número de unidades que indica el valor que está a continuación". El valor que sigue a la primitiva en el primer caso es 60, por eso la tortuga avanza 60 unidades; pero fíjese que LOGO no admite las siguientes instrucciones:

```
?fd sesenta
I don't know how to sesenta
?fd "sesenta
fd doesn't like sesenta as input
?fd :sesenta
sesenta has no value
?
```

Las razones que hay detrás de cada una de las respuestas son las siguientes:

— **sesenta** Se supone que es un procedimiento o una primitiva, dado que no va precedido de comillas ni de dos puntos, pero el intérprete de LOGO no encuentra tal procedimiento en su lista.

— **"sesenta** Se supone que es un nombre porque va precedido de comillas, pero un nombre no es un valor a no ser que se trate de un número.

— **:sesenta** Se supone que se que se refiere al valor de un objeto conocido como **sesenta** porque va precedido de dos puntos, pero, como tal objeto no existe, no puede tomar ningún valor.

Más adelante, en varios puntos de la Guía veremos con más detalle todas estas distinciones.

¿Qué sucedería si no se colocase ningún parámetro a continuación de **fd**? Averígüelo:

```
?fd
Not enough inputs to fd
?
```

Está claro que LOGO espera algún valor a continuación de **fd**, y en este caso no se ha incluido ninguno. LOGO emplea el término "**inputs**" en lugar de la expresión "**valores que deben seguir a ...**", que aunque es más larga no deja ninguna duda sobre su significado. El intérprete de LOGO espera un parámetro a continuación de **fd** y no ha recibido ninguno.

Si usted no intentó escribir el procedimiento:

```
?to estrella :lado :angulo :veces
```

en el capítulo 3, éste es un buen momento para hacerlo. Recuerde que no tiene que preocuparse de cometer errores; usted puede y debe aprender algo de ellos.

4.3 Nombres, objetos y procedimientos

Recuerde que nuestro propósito era ampliar el procedimiento **estrella** para que emplease tres parámetros, lo que permitiría cambiar el número de veces que LOGO repite la lista de instrucciones. Recuerde también que la lista de instrucciones era **[fd :lado rt :angulo]**, e iba precedida de **repeat 200**, en la que **200** representa el número de veces que se repite la lista de instrucciones.

La solución es muy sencilla. Basta con escribir:

```
?to estrella :lado :angulo :veces
>repeat veces [fd :lado rt :angulo]
>end
estrella defined
?
```

Ahora, para hacer que **estrella** dibuje un cuadrado de 100 unidades de lado no tiene más que dar la instrucción:

```
?estrella 100 90 4
I don't know how to veces in estrella: repeat veces
[fd :lado rt :angulo]
?
```

con lo que observará que ha cometido algún error al definir **estrella**.

Para facilitar un poco la comprensión de este error, vamos a detenernos un momento a investigar cuáles son las diferencias entre el nombre de un objeto (por ejemplo **yo**), el valor contenido en el objeto (**programador**) y el nombre de un procedimiento. Para iniciar nuestro análisis escriba las siguientes instrucciones:

```
?"yo
yo
?yo
I don't know how to yo
?:yo
yo has no value
?make "yo "programador
?:yo
programador
?make :yo "estupendo
?:programador
estupendo
?
```

En este ejemplo hay siete líneas de instrucciones que comienzan con el símbolo "**?**". Vamos a estudiarlas en el orden en que aparecen:

1. La palabra **"yo** se distingue por ir precedida de comillas. Cuando una secuencia de caracteres que no contenga espacios intercalados vaya precedida por el símbolo ", el ordenador considera el resto de caracteres como una palabra o unidad. Una palabra no es más que una secuencia de caracteres sin espacios intercalados, así que basta con colocar delante el símbolo " para definir su estado. Cuando escribe **"yo**, LOGO trata la secuencia como si fuera la palabra **yo** y por tanto escribe en pantalla precisamente **yo**.

2. Si escribe **yo** sin ningún signo delante, LOGO considera la secuencia de caracteres como el nombre de un procedimiento. Para hacer que LOGO realice un procedimiento no tiene más que darle el nombre del mismo, como vimos al trabajar con **estrella** o con **fd**. Este proceso también se denomina "llamar" al procedimiento o primitiva correspondiente. En resumen, enviar a LOGO una secuencia de caracteres que no vaya precedida por : o por " (véase el tercer apartado) requiere que exista un procedimiento o primitiva con el mismo nombre, de lo contrario LOGO responde "No sé cómo hacer ..."

3. Cuando escribe una palabra precedida de dos puntos (:), LOGO considera que nos referimos al "valor" de un objeto. En este caso el nombre del objeto es **yo** y por tanto la instrucción **:yo** solicita a LOGO su valor; pero hay que tener en cuenta que **yo** todavía no tiene valor.

4. A continuación se ha dado a **yo** el valor **programador**. Ambas palabras tienen que ir precedidas de comillas, para que LOGO sepa que estamos hablando de palabras y no de procedimientos. En la próxima línea se comprueba si se ha realizado la asignación correctamente.

5. Se solicita otra vez a LOGO el valor del objeto **yo**; para ello no hay más que darle el nombre de dicho objeto. LOGO confirma que en este momento el valor de **yo** es **programador**. Tenga presente este ejemplo al examinar las siguientes líneas.

6. Finalmente se ha dado a **yo** el valor **estupendo** (precedido de comillas, como siempre, para evitar que LOGO piense que se trata de un procedimiento). Como el valor anterior de **yo** era **programador**, esta última instrucción ha hecho que **programador** tome el valor **estupendo**. Conceptualmente podría sustituirse **make :yo "estupendo** por su equivalente **make "programador "estupendo**.

Para comprobar esta última afirmación no tiene más que solicitar a LOGO el valor de **programador**, por medio de **:programador**, para ver que su valor es **estupendo**.

Ahora ya podemos volver a la **estrella** que no habíamos conseguido definir correctamente.

4.4 Otra estrella nueva

La anterior definición fue la siguiente:

```
?po "estrella
to estrella :lado :angulo :veces
repeat veces [fd :lado rt :angulo]
end
?estrella 100 90 4
I don't know how to veces in estrella: repeat veces
[fd :lado rt :angulo]
?ed
```

y se había quedado en modo de edición con el cursor situado precisamente al comienzo de **veces**, es decir, en el lugar en que LOGO había localizado el error.

Como LOGO espera que **repeat** vaya seguido de un valor y no le hemos dado un número de forma explícita, es preciso que el objeto que sigue a la primitiva tenga un valor (que identificaremos empleando :) o que se trate de un procedimiento que a su vez nos proporcione el parámetro que necesitamos. Nosotros no hemos definido **veces** como procedimiento, y realmente lo que tendríamos que haber escrito no es **veces**, sino **:veces**. Como nos encontramos en el editor es muy sencillo realizar las correcciones necesarias para que el procedimiento quede de la siguiente forma:

```
to estrella :lado :angulo :veces
repeat :veces [fd :lado rt :angulo]
end
```

con lo que finalmente todo funciona como es debido. Pruebe con unos cuantos valores.

Se puede cometer muchos otros errores, pero la mayor parte serán fallos de mecanografía. Por ejemplo, ¿cuál cree que es el error en la definición del siguiente procedimiento?

```
?estrella 100 90 4
vces has no value in estrella :repeat :vces
[fd :lado rt :angulo]
?po "estrella
to estrella :lado :angulo :veces
repeat ;vces [fd :lado rt :angulo]
end
?
```

Sugerencia: compruebe cómo se ha escrito cada nombre.

Por último trate de hallar el error de la siguiente línea:

```
?to estrella :lado :angulo veces
veces isn't a parameter
?
```

Esta vez no voy a dar ninguna pista.

Capítulo 5

Programas aleatorios

En este capítulo vamos a practicar la utilización de procedimientos en LOGO, centrando nuestra atención en la tortuga.

Con los procedimientos definidos en este capítulo la tortuga viaja por su cuenta a donde quiere, controlada (valga la expresión) por los caprichos imprevisibles de un generador de números aleatorios. Vamos a dejar que la tortuga se mueva por su cuenta y riesgo para centrarnos en el diseño de una serie de posibilidades un tanto más restringidas.

Uno de los objetivos más importantes de este capítulo es presentar los conceptos de "secuencia", "repetición" y "control", así como tomar contacto con la noción de "recurrencia", que nos va a acercar a una de las herramientas más poderosas de que dispone el programador de LOGO. Si consigue hacerse con el concepto de "recurrencia" habrá dado un buen paso adelante a la hora de comprender, por ejemplo, muchas de las ideas de la inteligencia artificial.

La recurrencia es la característica que permite llamar a un procedimiento desde otro procedimiento del mismo tipo, intercalándose en éste de la misma manera que en el pensamiento unas ideas se intercalan en otras, o de la misma forma que en el lenguaje humano unas oraciones se intercalan en otras. La inteligencia humana tiene mucho que ver con las formas de emplear el pensamiento y el lenguaje. Para que un ordenador pueda comprender al menos parte de nuestros pensamientos y de nuestro lenguaje, es necesario que esté dotado de un lenguaje informático que permita intercalar unos procedimientos en otros, es decir, un lenguaje que admita procedimientos recurrentes. LOGO es un lenguaje de este tipo.

5.1 Generación de números aleatorios

En muchos juegos para ordenadores, asicomo en muchos juegos tradicionales, el resultado depende en gran medida de los conceptos "suerte" y "casualidad". A lo largo de la historia de las matemáticas los hombres hemos desarrrollado toda una ciencia, la estadística, y uno de sus campos de aplicación, la probabilidad, para investigar los fenómenos que pueden suceder o que han sucedido "por casualidad". Pero ¿qué es un fenómeno aleatorio o casual?

Vamos a definir el procedimiento **mover**:

```
?to mover
>fd 15
>end
mover defined
?
```

Si llama a este procedimiento verá que la tortuga avanza una longitud casi insignificante. Sin embargo, si las llamadas son varias conseguirá que la tortuga avance más.

Ahora escriba las siguientes líneas:

```
?random 360
239
?random 360
86
?random 360
344
?random 360
29
?
```

Probablemente los resultados que usted obtenga coincidirán con los del texto, a no ser que ya haya empleado el generador de números aleatorios. Los números aleatorios que genera Dr LOGO siguen una secuencia establecida previamente, así que para obtener una serie que comience en un valor desconocido hay que comenzar en un punto también aleatorio.

Para ver la regularidad del generador de números aleatorios, vuelva a inicializar LOGO dando la orden **bye**. Una vez que se encuentre en CP/M, cargue de nuevo LOGO y escriba la siguiente línea:

```
?repeat 10 [pr random 360]
239
86
344
29
140
36
218
79
134
265
?
```

Verá que los cuatro primeros números son los mismos que en el ejemplo anterior. Se puede partir de un origen aleatorio empleando el procedimiento **randomize**, con el que volveremos a trabajar un poco más adelante en el capítulo "Nombres y propiedades". Este procedimiento emplea la primitiva **nodes** para proporcionar un número aleatorio como punto inicial.

```
?to randomize
>repeat nodes - 50 * int (nodes / 50) [if 2 =
random 2 []]
>end
randomize defined
?randomize
?
```

El procedimiento **randomize** que acabamos de definir comienza la secuencia con un valor imprevisible, ya que la lista de instrucciones se repite un número de veces indeterminado. Esta lista no realiza ninguna acción debido a que cuenta con otra lista vacía []. Utilizando varios procedimientos que veremos en el capítulo "Nombres y propiedades", he estudiado el comportamiento del generador de números aleatorios y he podido comprobar que dicho comportamiento es realmente aleatorio.

5.2 Una tortuga vacilante

Compare cómo afectan estos dos grupos de instrucciones a la orientación de la tortuga. En primer lugar escriba:

```
?rt 45
?lt 45
?
```

y a continuación:

```
?rt 45
?rt 315
?
```

Ambos resultados son iguales. La tortuga gira 45 grados a la derecha para volver a tomar a continuación la orientación de partida. Para la tortuga la instrucción **lt 45** es equivalente a **rt 315**. Esto es debido a que 45 más 315 suman 360 grados, es decir, una circunferencia completa.

Ahora escriba:

```
?mover
I don't know how to mover
?to mover
>fd 15
>end
mover defined
?
```

puesto que al salir de LOGO por medio de **bye** se perdió la primera definición del procedimiento **mover**. A continuación escriba:

```
?rt random 360 mover rt random 360 mover
?rt random 360 mover rt random 360 mover
?rt random 360 mover rt random 360 mover
?rt random 360 mover rt random 360 mover
?
```

Para repetir las líneas no es preciso volver a introducirlas de nuevo; puede emplear CTRL Y. Una vez que haya repetido esta línea unas cuantas veces, aunque use CTRL Y empezará a preguntarse si no existe un método más sencillo. Con LOGO siempre suele haberlo.

Si el procedimiento **mover** simplifica bastante el avance de la tortuga, ¿por qué no crea un procedimiento que simplifique también los giros? Lo único que hay que hacer es sustituir **rt random 360** por una sola palabra, es decir, por un procedimiento al que daremos un nombre adecuado, como por ejemplo **vacilar**. Dejo que sea usted quien defina el procedimiento que se encarga de hacer girar a la tortuga de forma que tome direcciones aleatorias. Aquí tiene una pista: comience con

```
?to vacilar
```

Ahora, puede comprobar que el procedimiento que usted ha definido funciona correctamente empleándolo en el siguiente ejemplo:

```
?cs
?vacilar mover vacilar mover vacilar mover
?
```

En la segunda línea aparece varias veces el procedimiento **vacilar**. Si todo ha funcionado correctamente, ¿por qué no emplear la sentencia **repeat**?

```
?to paseo_corto
>repeat 200 [vacilar mover]
>end
paseo_corto defined
?
```

En este ejemplo se ha definido otro procedimiento que se encarga de hacer que la tortuga de un **paseo_corto**. Al ejecutar este procedimiento la tortuga parece vacilar un rato hasta pararse. Puede volver a ejecutar este procedimiento y pulsar ESC antes de que acabe, con lo que se encontrará con el siguiente mensaje:

Stopped! in paseo_corto: mover

o tal vez con este otro:

Stopped! in paseo_corto: vacilar

o con alguno similar dependiendo del punto en que haya interrumpido la ejecución del procedimiento. En caso de que quiera modificar el número de repeticiones, puede modificar el parámetro que sigue a **repeat** hasta un valor máximo de:

```
?to paseo_largo
>repeat 32767 [vacilar mover]
>end
paseo_largo defined
?
```

Al dar un **paseo_largo** es bastante probable que en algún momento la tortuga se salga de la pantalla. Al entrar en LOGO desde CP/M el ordenador se encuentra en modo **window**, es decir, la pantalla forma parte de un área gráfica bastante más amplia. Para evitar que la tortuga desaparezca de la pantalla emplee la instrucción **wrap**.

```
?wrap fs cs paseo_largo
```

A continuación vamos a estudiar otro método con el que se puede obtener el mismo resultado de forma tal vez un poco más clara.

5.3 Un viaje más largo

En el apartado anterior hemos realizado un viaje (con **vacilar** y **mover**), a continuación otro viaje (también con **vacilar** y **mover**) y luego otro más (otra vez con los mismos procedimientos) y después ... Necesitamos un procedimiento, al que llamaremos **viajar**, que se componga de **vacilar**, **mover** y por último otra vez de **viajar**. El procedimiento se define así:

```
?to viajar
>vacilar mover viajar
>end
viajar defined
?
```

Si ejecuta este procedimiento, verá que la tortuga comienza a pasearse y sigue así hasta que usted se canse e interrumpa el procedimiento, ¿no es así?

El procedimiento **viajar** es lo que se denomina un procedimiento "recurrente", porque vuelve a llamarse a sí mismo. Siempre que LOGO llama a un procedimiento, por ejemplo a **vacilar**, podemos considerar que sustituye el nombre del mismo por su definición:

```
?po "vacilar
 to vacilar
 rt random 360
 end
?
```

lo que quiere decir que también podría haberse definido **viajar** con **rt random 360 mover viajar**. Observe que acabo de dar la definición de **vacilar** que antes omití. También podría sustituirse el procedimiento **mover** por **fd 15**, con lo que **viajar** quedaría como **rt random 360 fd 15 viajar**.

Si seguimos sustituyendo todos los procedimientos a los que llama **viajar** por su definición, obtendríamos finalmente:

```
rt random 360 fd 15 rt random 360 fd 15 rt random
...
```

lo que significa que **viajar** es una lista indefinida de instrucciones que en la práctica viene limitada únicamente por las capacidades de LOGO.

El tipo de recurrencia que hemos empleado en **viajar** se denomina "cola", dado que la lista de instrucciones forma una cola cada vez más larga. Dr LOGO puede ejecutar un procedimiento recurrente de cola durante bastante tiempo, siempre y cuando el procedimiento no solicite datos cada vez. De hecho he puesto a la tortuga a **viajar** durante periodos bastante largos, con la pantalla en modo **wrap**; la huella de la tortuga ha dejado la pantalla casi totalmente cubierta.

5.4 Cómo controlar el movimiento

La única forma de detener los paseos de la tortuga en el anterior procedimiento es pulsar las teclas [ESC] o [CTRL][G], pero si quiere emplear un método más sofisticado puede probar con:

```
?to viajar?
>vacilar mover
>if not keyp [viajar?]
>end
viajar? defined
?cs viajar?
```

Con este sistema la tortuga empieza a pasearse por la pantalla hasta que pulse cualquier tecla que no sea ESC. El único problema es que la tecla pulsada se reproduce en la pantalla para textos.

La primitiva **keyp** comprueba si se ha pulsado cualquier tecla, dando como resultado **TRUE** (cierto) o **FALSE** (falso). Se puede pasar de **TRUE** a **FALSE** y a la inversa por medio de **not**, como en el siguiente ejemplo:

```
?not TRUE
I don't know how to TRUE
?not "TRUE
FALSE
?not "true
true is not TRUE or FALSE
?
```

El último mensaje de LOGO (“**true is not TRUE or FALSE**”, es decir, “lo verdadero no es VERDADERO ni FALSO”) parece tener un profundo contenido filosófico. En la práctica, lo único que quiere decir es que para emplear estos términos hay que escribirlos en mayúsculas y precedidos por unas comillas.

Para resolver el problema anterior tendríamos que encontrar una forma para que la tecla pulsada no apareciese en pantalla; la solución es bastante sencilla:

```
?to viajar?
>vacilar mover
>if not keyp [viajar?] [borrainput]
>end
viajar? defined
?
```

La única novedad es la inclusión del procedimiento **borrainput** como acción alternativa en la sentencia **if**. En caso de que la condición se cumpla, LOGO vuelve a ejecutar **viajar?**, pero si es falsa pasa a **borrainput**. No es necesario definir **borrainput** antes de definir **viajar?**, así que todavía podemos pensar tranquilamente qué es lo que queremos que realice este procedimiento.

Realmente lo que queremos es que LOGO lea la tecla pulsada en el teclado, pero que no haga nada con esta información. Hay varios métodos que permiten ver si se ha pulsado alguna tecla, por ejemplo:

```
?rl
```

seguido de:

```
hj jj ll INTRO
[hj jj ll]
```

y de:

```
?rq
hj jj ll INTRO
hj jj ll
?count rl
1 2 3 INTRO
3
?count rq
1 2 3 INTRO
5
?rc
m
?count rc
1
?
```

LOGO cuenta con tres primitivas que le permiten leer la información escrita:

1. **rl**, abreviatura de "readlist" (leer lista). Toma todo lo que escribimos hasta el primer INTRO y lo convierte en una lista. De esta forma **hj ii ll** pasa a ser **[hj ii ll]** y **1 2 3** pasa a ser **[1 2 3]**, que es una lista de tres elementos.
2. **rq**, abreviatura de "readquote" (leer literalmente). Esta instrucción vuelve a tomar la misma línea que en el caso anterior y la convierte en una palabra. De esta forma **1 2 3** pasa a ser **"1 2 3**, que tiene cinco elementos dado que los espacios intercalados también cuentan.
3. **rc**, abreviatura de "readcharacter" (leer carácter). Lee un carácter en cuanto lo escribimos, pero no lo reproduce en pantalla; la **m** que vemos es la respuesta de **rc**, y no proviene del teclado como es habitual. Para comprobarlo no hay más que ver que la respuesta de **count rc** es **1**, siendo **m** el carácter pulsado.

Una vez vistas las características de estas primitivas de lectura, está claro que para nuestros fines nos conviene emplear **rc**.

Además tenemos que hacer algo con el carácter, así que podemos asignarlo a una variable local.

```
?to borrainput
>local "c
>make "c rc
>end
borrainput defined
?
```

De esta forma hemos conseguido una fórmula bastante más elegante para detener los paseos de la tortuga.

Compruebe que funciona realmente.

Capítulo 6

La tortuga viajera y el color

En el capítulo anterior no hemos dibujado ninguna figura ni ningún diagrama, dado que aún no hemos visto cómo ponerlas color. Ahora vamos a resarcirnos plenamente con una serie de procedimientos que nos permiten trabajar con los colores, empleando fundamentalmente las capacidades de los ordenadores Amstrad y no posibilidades adicionales que aporte LOGO.

Una vez que hayamos visto estas técnicas, su aplicación a muchos otros campos queda ya en manos de cada uno.

6.1 Códigos de control

En el capítulo de "Referencias del Manual del Usuario de Amstrad" encontrará un apartado sobre caracteres de control a los que habitualmente no puede acceder por medio de la tecla [CTRL].

Si introduce:

```
?[CTRL][G]
Stopped!
?char 7

?
```

verá que la respuesta del ordenador a la instrucción **char 7** es una línea en blanco junto con la emisión de un pitido. Si escribe **char 2** verá que el cursor desaparece de repente, volviendo a aparecer en cuanto introduzca **char 3**. Todos estos fenómenos que pueden parecer bastante extraños vienen perfectamente explicados en el capítulo de Referencias que acabamos de citar.

A continuación está reproducida una parte de la tabla de códigos de control.

Valor	Nombre	Parámetros	Significado
&00 0	NUL		Sin efecto. LOGO la ignora.
&02 2	STX		Hace invisible el cursor.
&03 3	ETX		Vuelve a hacer visible el cursor.
&04 4	EOT	0 .. 2	Establece el modo de pantalla.
&07 7	BEL		Produce un pitido.
&1C 28	FS	0 .. 15	Asigna un par de colores a una tinta.
		0 .. 31	Equivale a la orden **INK.**
&1D 29	GS	0 .. 31	Asigna un par de colores al borde.
		0 .. 31	Equivale a la orden **BORDER.**

La primitiva **char** guarda una estrecha relación con la tabla anterior. Vamos a emplear los caracteres de control para modificar a nuestro gusto las salidas por pantalla. Su uso es muy sencillo. Comencemos con un procedimiento que permite fijar el color del borde:

```
?to colorborde :valor
>make "valor + 32
>op (word char 29 :valor char :valor)
>end
colorborde defined
?colorborde 13

?
```

con lo que la pantalla queda rodeada de un borde blanco. Si quiere un borde que parpadee escriba:

```
?(word char 29 char 2 char 18)

?
```

También puede conseguir una mezcla realmente horrorosa con:

```
?(word char 29 char 6 char 18)

?
```

Vuelva a una situación más tranquila con **colorborde 13.**

El parámetro de **colorborde, :valor,** corresponde a los números del ''MASTER COLOUR CHART'' del Amstrad. En este cuadro puede ver que el número 13 corresponde al color blanco. En la siguiente línea de la definición del procedimiento hemos aumentado el valor de este parámetro a 32 unidades; el valor real que emplea

el procedimiento es el resto de una división por 32, y de esta forma se evita la emisión del carácter de control 0, correspondiente a NUL, que detiene el funcionamiento regular.

La siguiente línea emplea la primitiva **word** para enviar al ordenador una secuencia de caracteres que no incluya espacios ni retornos de carro. Esta instrucción va entre paréntesis junto con sus parámetros, dado que admite un número de datos variable. La secuencia final enviada al ordenador se compone del carácter de control 29 seguido de un par de parámetros que representan los colores del borde. Si los dos últimos parámetros son distintos, el borde parpadea.

Es conveniente comparar el procedimiento anterior con la descripción de los caracteres de control que aparece en el Manual del Usuario.

6.2 Algo más sobre los colores

Una vez que hemos conseguido modificar el color del borde, vamos a intentar cambiar el del papel. En el LOGO de Amstrad el color inicial del papel siempre es 0, así que debe haber bastantes similitudes entre el procedimiento **colorpapel** y el procedimiento **colortinta**, que lógicamente habrá que definir con anterioridad. Estos procedimientos son los siguientes:

```
?to colortinta :tinta :color
>make "tinta :tinta + 16
>make "color :color + 32
>op (word char 28 char :tinta char :color char :color)
>end
colortinta defined
?to colorpapel :valor
>op colortinta 0 :valor
>end
colorpapel defined
?colorpapel 0 colortinta 1 18

?colortinta 2 6 colortinta 3 2

?
```

Hemos hecho que el papel tome color negro y la pluma verde brillante; de momento la pluma está empleando tinta 1. Si da la orden **cs**, borrado de pantalla, aparece la tortuga en verde. Si ahora ejecuta **setpc 2**, la tortuga aparece en rojo brillante; del mismo modo, si ejecuta **setpc 3** la verá en azul brillante. La primitiva **setpc** determina el color de la tinta, es decir, lo que en nuestra jerga denominamos color de la pluma.

Dr LOGO cuenta con una primitiva especial que le permite definir los colores: **setpal**.

Esta primitiva emplea dos parámetros; el primero es el número de la tinta, mientras que el segundo es una lista con las intensidades de los tres colores básicos en el siguiente orden: rojo, verde y azul. Estos tres colores constituyen el sistema RGB que emplean la mayor parte de los monitores de color. Cada uno de los tres colores puede tomar tres intensidades: 0 (sin color), 1 (normal) y 2 (brillante).

Por ejemplo, para que la tinta 3 tome un color azul normal no hay más que ejecutar **setpal 3 [0 0 1]**; para dar a la tinta 2 una combinación de rojo y verde, ejecute **setpal 2 [1 1 0]**, con lo que obtendrá un amarillo. Por último, si quiere que la tinta 1 sea una combinación de rojo brillante y azul brillante ejecute **setpal 1 [2 0 2]**, con lo que se obtiene un magenta brillante.

Antes de modificar los colores de las tintas intente realizar este pequeño ejercicio:

```
?cs
?setpal 1 [0 2 0]
?setpc 1
?fd 150 rt 120
?setpal 2 [2 0 0]
?setpc 2
?fd 150 rt 120
?setpal 3 [0 0 2]
?setpc 3
?fd 150
?
```

Si ahora escribe:

```
setpal 0 [0 0 0]
```

y mueve la tortuga, verá que dibuja una línea invisible, dado que ha hecho que el color de la tinta sea igual que el del papel.

Los procedimientos **colortinta** y **colorpapel** le permiten emplear los números de los colores a los que está acostumbrado, pero la primitiva **setpal** le abre un campo de experimentos mucho más interesante.

Fíjese, sin embargo, que el LOGO de Amstrad no cuenta con una primitiva que permita cambiar el color del borde.

6.3 ¿El viaje final?

Las figuras 6.1 y 6.2 muestran dos ejemplos de viajes a todo color.

La primera figura presenta en pantalla un dibujo bastante pequeño, debajo del cual podemos leer:

```
Out of LOGO stack space
?
```

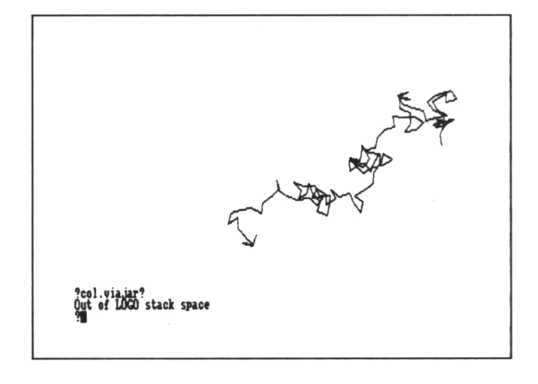



Fig. 6.1

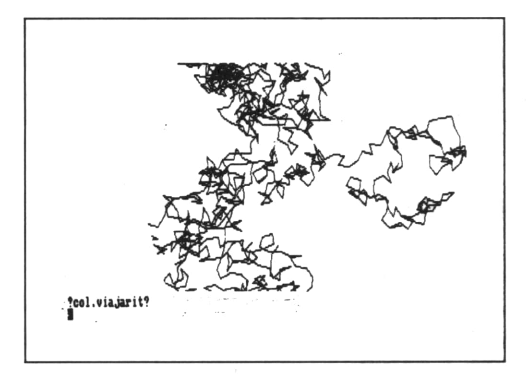


Fig. 6.2

que significa que LOGO no tiene suficiente memoria para recordar el procedimiento que se encarga del dibujo. Esto es algo que no había ocurrido con **viajar?**. Recuerde que habíamos definido el procedimiento de la siguiente forma:

```
?to col.viajar?
>vacilar mover
>setpc 1 + random 3
>if not keyp [col.viajar?] [borrainput]
>end
col.viajar? defined
?
```

Se trata de un procedimiento que ya le es familiar; la única modificación que se ha introducido ha sido incorporar cambios aleatorios en la tinta, pero al conservar en una parte de la memoria que se denomina "pila" (del inglés "stack") todos los cambios de color, en seguida se agota la memoria. Aunque el concepto de recurrencia permite una elegancia conceptual en los programas, en algunas circunstancias da lugar a situaciones prácticas bastante incómodas.

Para obtener la segunda figura no se ha empleado el procedimiento recurrente, sino

un procedimiento iterativo con el que puede hacerse que la tortuga se mueva durante bastante más tiempo. El procedimiento iterativo es el siguiente:

```
?to col.viajarit?
>repeat 32767 [vacilar mover setpc 1 + random 3
if keyp [borrainput stop]]
>end
col.viajarit? defined
?
```

Hay que destacar dos cosas en este procedimiento: el uso de **stop** para terminar la ejecución del mismo y lo difícil que resulta escribir **col.viajarit?** sin equivocarse.

En algunos ambientes informáticos se desprecia el uso de los métodos iterativos frente a los recurrentes, que se consideran superiores en todos los aspectos de una forma casi mística. La recurrencia no tiene por qué ser superior en todo momento a la iteración (que puede definirse como todo aquello que emplea algún tipo de elemento que se repite). Cada cosa tiene su propio lugar ...

6.4 Otros medios para mover la tortuga

Hay un método bastante elegante para mover la tortuga por la pantalla que consiste en utilizar un joystick. A continuación vamos a estudiar detalladamente varios recursos de que disponemos en LOGO; en particular vamos a investigar los misterios de las primitivas **throw** y **catch**.

Comencemos con unas cuantas cosas bastante sencillas.

El primer procedimiento es:

```
?to tortuga
>repeat 32767 [girar mover]
>end
tortuga defined
?
```

que realmente emplea un bucle para repetir los efectos de **[girar mover]**. Este procedimiento tiene un poco de todo. En 32767 ocasiones vemos a la tortuga **girar** y **mover**se hacia delante, aunque todavía no sabemos cómo vamos a hacer que nuestra "pequeña bestia" gire y avance.

El primero de los procedimientos que necesitamos es:

```
?to mover
>fd 15
>end
mover defined
?
```

Este procedimiento se encarga de hacer que la tortuga avance 15 unidades cada vez que le llamamos. Es la historia de siempre. La novedad aparece en el segundo procedimiento, **girar**:

```
?to girar
>rt (45 *joy)
>end
girar defined
?
```

Para comprender cómo funciona este procedimiento tenemos que ver previamente cómo funciona un joystick. El joystick que voy a describir es el de Amstrad, pero he trabajado con muchos otros que funcionan exactamente igual.

El joystick no es más que una palanca negra que se mueve y que tiene un botón rojo en su parte superior. Si coge correctamente el joystick, verá la palabra "TOP" en la parte delantera del cuerpo del instrumento. Conecte el joystick a la parte posterior del ordenador e introduzca el siguiente procedimiento:

```
?to leerjoy
>repeat 32767 [pr paddle 0 if buttonp 0
[stop]]
>end
leerjoy defined
?
```

Si gira la palanca del joystick mientras ejecuta el procedimiento leerjoy, verá aparecer en pantalla números que van del 0 al 7, y frecuentemente aparecerá también el número 255. Pruebe con distintas posiciones del joystick para ver qué valores aparecen en la pantalla. Para salir de este procedimiento pulse el botón del joystick.

Si usted cuenta con dos joysticks tendrá que cambiar la lista de instrucciones, poniendo **paddle 0 paddle 1 if buttonp 0 or buttonp 1** [... para investigar el funcionamiento de cada uno de ellos. El apéndice A indica cuál es la relación que hay entre los números que aparecen en pantalla y la posición de la palanca. Para ello se ha empleado el procedimiento **dibujoy**. Es necesario definir este procedimiento con el editor, dado que la lista de órdenes que se encargan de mover la tortuga es muy larga.

```
?to dibujoy
>local "val
>cs fs ht
>make "val 0
>repeat 8 [pu fd 30 pd fd 110 pu fd 10 tt :val
bk 150 rt 45 pd make "val :val + 1]
```

```
>pu fd 6 lt 90 fd 22
>tt "255 rt 90
>fd 180 lt 90 20 tt "ARRIBA
>end
dibujoy defined
?
```

Aparte de la pequeña complejidad que representa la ubicación de la tortuga, lo más importante de este procedimiento es el uso de la primitiva **tt** (turtle text), que permite escribir en la pantalla el valor del dato que la sigue.

6.5 Valores procedentes de un procedimiento

El procedimiento **girar** sólo contiene una línea activa: **rt (45 * joy)**. Para que LOGO pueda ejecutar esta línea es preciso que **joy** tenga un valor, en cuyo caso la tortuga gira a la derecha el número de grados que resulta de multiplicar 45 por el valor de **joy**.

joy es un procedimiento, y hasta ahora hemos visto que los procedimientos realizan acciones pero no producen ningún valor. Para que un procedimiento produzca un valor se usa la primitiva **op**. La definición de **joy** es la siguiente:

```
?to joy
>local "joyval
>make "joyval paddle 0
>if :joyval = 255 [op 0] [op :joyval]
>end
joy defined
?
```

Ahora para mover la tortuga no hay más que ejecutar el procedimiento **tortuga**. También se la puede mover con el teclado numérico, pero de eso trataremos más adelante.

6.6 Textos y gráficos

Ya hemos visto que con la primitiva **tt** se puede escribir palabras en la pantalla de gráficos. Los dos procedimientos siguientes permiten poner rótulos a los dibujos:

```
?to rotular
>local "ps
>make "ps item 4 tf
>pu
>catch "ROTULO [repeat 32767 [leertecla]]
```

```
>if :ps = "PD [pd]
>recycle
>end
?to leertecla
>local "ck
>make "ck rc
>if :ck = char 56 [seth 0 fd 16]
>if :ck = char 50 [seth 180 fd 16]
>if :ck = char 52 [seth 270 fd 16]
>if :ck = char 54 [seth 90 fd 16]
>if :ck = char 53 [tt rq]
>if :ck = char 32 [throw "ROTULO]
>end
leertecla defined
?
```

Para facilitar un poco la comprensión del segundo procedimiento, **leertecla**, aquí tiene algunas equivalencias entre caracteres y sus valores ASCII:

CARACTER	ASCII	FUNCION
" (espacio)	32	Parar
"2	50	Hacia abajo
"4	52	A la izquierda
"5	53	Escribir rótulos
"6	54	A la derecha
"8	56	Hacia arriba

El procedimiento **leertecla** espera hasta que es pulseda una tecla y asigna el carácter correspondiente a la variable local **ck**. La serie de comparaciones múltiples, es decir, los **if** sucesivos, tienen por objeto comprobar si la tecla pulsada es una de las que reconoce el sistema.

Hemos identificado las teclas por sus códigos ASCII para distinguir más fácilmente los caracteres numéricos y sus códigos, por ejemplo "2 y el código 50. Si comparamos el teclado numérico con las direcciones correspondientes podremos ver si el usuario está empleando el teclado numérico para mover el cursor.

En la tabla anterior, en el centro, nos encontramos con el carácter **"5**, que es el que se emplea para activar la secuencia **tt rq**. Para finalizar la rotulación no hay más que pulsar la barra espaciadora; fíjese que se ha elegido una tecla totalmente distinta de las del teclado numérico para evitar confusiones. Para salir del procedimiento **leertecla** se emplea **throw "ROTULO**; por otra parte, **catch "ROTULO** forma parte del procedimiento **rotular**.

El procedimiento **rotular** empieza con una variable local que contiene el estado de la pluma, es decir, **item 4 tf**. Asignamos este valor a **ps** para emplearlo más tarde cuando queramos modificar el estado de la pluma. Tenga en cuenta que es necesa-

rio levantar la pluma para ir de un rótulo a otro, de lo contrario la tortuga iría dejando huella sobre el dibujo.

El procedimiento **leertecla** se ejecuta hasta 32767 veces dentro de **catch "ROTULO**. A continuación el control pasa a la siguiente línea; en caso de que el estado inicial de la pluma fuese **PD**, vuelve a bajarse. Finaliza el procedimiento con una instrucción **recycle**, que organiza un poco la memoria, atando cabos sueltos, por si queremos volver a utilizar el procedimiento **rotular**.

Y ahora vamos con los números.

Capítulo 7

Sonido

En los ordenadores Amstrad, el sonido en LOGO tiene muchas similitudes con el sonido en BASIC.

Vamos a estudiar por separado los formatos de las primitivas relacionadas con el sonido.

7.1 Sound

Esta orden tiene el siguiente formato:

sound (estado de canal, tono, duración, volumen, envolvente de volumen, envolvente de tono, ruido)

A continuación veremos qué representa cada uno de los parámetros de esta orden.

7.1.1 Estado del canal

El estado del canal es un número que se determina sumando otros números que, a su vez, representan dónde ha de interpretarse el sonido. Los ordenadores Amstrad tienen tres canales, a los que llamaremos A, B y C, que se pueden programar por separado. Si no incluimos este parámetro, como por ejemplo:

```
?sound [ ]
?
```

oiremos una especie de chasquido en el canal B. Normalmente siempre debe incluirse este parámetro.

La primitiva **sound** controla varias acciones que pueden reducirse a dos principales, enviar el sonido y sincronizar los canales, y a dos complementarias.

La primera acción de **sound**, enviar el sonido a los canales, activa uno o varios canales. Cada canal está representado por un número binario que lo identifica, y que es 001 para el canal A, 010 para el B y 100 para el C (1, 2 y 4 en decimal, respectivamente).

La otra acción principal que realiza esta primitiva es sincronizar los canales. Esto quiere decir que **sound** puede ordenar a varios canales que esperen hasta que se produzca un hecho determinado; por ejemplo, puede hacer que el canal A se sincronice con el B. A partir de ese momento, las instrucciones **sound** que enviemos al canal A pasarán a formar parte de una cola y no se ejecutarán hasta que se sincronicen ambos canales. Mientras tanto el canal B puede estar interpretando cualquier sonido, dado que la sincronización no se ejecuta hasta que el canal B recibe también una orden para sincronizarse con el A.

También es posible solicitar una sincronización múltiple, pero en este caso hay que tener en cuenta que las instrucciones se mantienen en la cola y que cada sincronización no se ejecuta hasta que se recibe la otra correspondiente o hasta que se completa la instrucción **sound**. En este caso los números correspondientes también se basan en la numeración binaria. El número a sumar para sincronizar con el canal A es 8, 16 para el B y 32 para el C.

Si añadimos 64 a este parámetro conseguiremos "bloquear" la cola de sonidos; esto quiere decir que los canales especificados en la instrucción **sound** dejan de producir sonidos. Si el ordenador encuentra esta instrucción en la cola, también bloquea todas las instrucciones de sonido que aparecen a continuación de la misma.

Por ejemplo, supongamos que enviamos tres instrucciones **sound** consecutivas, en las que el valor del estado del canal es respectivamente 1, 67 y 2. En primer lugar suena el canal A. A continuación el ordenador bloquea los canales A y B (fíjese que 67 es 1 + 2 + 64). Estos canales continúan bloqueados hasta que los liberemos con **release**, momento en el cual sonarán ambos simultáneamente. Finalmente, cuando hayan terminado ambos volverá a sonar el canal B. Sin preocuparnos demasiado por el significado de la instrucción, escriba:

```
?sound [1 200 200] sound [67 300 100] sound
[2 100 200]
?
```

y oirá que el altavoz emite un sonido de 125000/200 hertzios (es decir, de 625 hertzios) durante 200 centésimas de segundo, es decir, durante 2 segundos.

Al terminar dicho sonido el ordenador no emitirá nada más hasta que escriba:

```
?release 3
?
```

que produce otro tono, esta vez de 125000/300 hertzios (416.7 hertzios) durante 100 centésimas de segundo, simultáneamente en los canales A y B. Una vez que haya emitido este sonido, el ordenador emitirá otro de 125000/100 hertzios (1250 hertzios) durante 200 centésimas por el canal B. Ha debido emplear **release 3** porque se trataba de liberar los canales A y B; 1 + 2 es 3.

El último valor que puede añadirse a este parámetro es 128, que corresponde al bo-

rrado de un canal. Para borrar el canal A tenemos que introducir **sound [129]**, puesto que 1 + 128 es 129. Pruebe con este ejemplo:

```
?sound [1 30 10000] sound [2 300 1000]
?sound [129]
```

si consigue introducir la segunda línea lo suficientemente deprisa, podrá apreciar que el sonido agudo termina, pero el grave continúa.

7.1.2 Tono

El tono viene representado por un número que varía entre 0 y 4095.

Habitualmente se toma como referencia un tono de 284, que equivale a 440 hertzios. Por convenio internacional esta frecuencia corresponde a la nota LA de la octava central, a la que se denomina LA internacional.

Se puede almacenar las notas correspondientes a la octava central, es decir, a la octava 0, en una variable local, a la que daremos el nombre **octava**, junto con su tono correspondiente. Esta variable será bastante útil en cualquier procedimiento que asocie los nombres de las notas con sus tonos correspondientes. Tenga en cuenta que este procedimiento debe escribirse con el editor debido a la longitud de la lista **octava**:

```
?to periodo :nota
>local "octava
>make "octava [DO 478 DO# 451 RE 426 RE# 402 MI 379
FA 358 FA # 338 SOL 319 SOL# 301 LA 284 LA# 268 SI 253]
>op buscar :octava :nota
>end
periodo defined
?
```

Este procedimiento acepta el nombre de una nota y lo compara con los contenidos de **octava**, produciendo el tono correspondiente.

El procedimiento **buscar** se encarga de realizar la búsqueda en la lista:

```
?to buscar :l :n
>if emptyp :l [op 0]
>if = :n first :l [op item 2 :l]
>op buscar bf :l :n
>end
buscar defined
?
```

En caso de que no encuentre el nombre de la nota, este procedimiento da como resultado el valor 0. Podemos emplear **periodo "DO** en lugar del valor 478 que corresponde a esta nota. Por ejemplo:

```
?periodo "DO
478
?periodo "MI#
0
?
```

Observe que el MI# no existe.

Si quisieramos emplear este procedimiento en una instrucción **sound** tendríamos que modificar ligeramente nuestro primer intento:

```
?sound [1 periodo "LA 200]
sound doesn't like [1 periodo "LA 200] as input
?sound (se 1 periodo "LA 200)
?
```

Debemos asegurarnos de que empleamos como parámetro de entrada una lista "activa", entendiendo como tal aquélla en que cada elemento se evalúa y ejecuta adecuadamente, para lo cual empleamos la primitiva **se**.

Observe que la ejecución de **periodo** lleva un cierto tiempo, que el ordenador emplea en calcular el tono. Esto tiene como consecuencia que la primitiva **sound** no se ejecute inmediatamente, sino que pasa a formar parte de la cola y se ejecuta cuando le llegua su turno. Compare las siguientes líneas:

```
?sound (se 1 periodo "LA 200)
?sound [1 100 200] sound (se 1 periodo "LA 200)
?
```

En el primer ejemplo hay que esperar un poco antes de oir el sonido, mientras que en el segundo caso el ordenador ejecuta inmediatamente el primer **sound**, ejecutando a continuación el segundo sin que se produzca ninguna pausa entre ambos.

Hay un ejercicio interesante que consiste en definir un procedimiento al que denominaremos **periodo :nota :numoctava**, que acepta una nota y el número de la octava a la que pertenece y da como resultado el tono correspondiente. Inténtelo.

7.1.3 Otros parámetros

La duración del sonido se mide en centésimas de segundo, y puede variar entre 0 y 32767.

El volumen varía entre 0 y 15. Si se omite este parámetro, LOGO toma implícitamente el valor 12.

La envolvente de volumen puede tomar un valor de 0 a 15, que representa el número de la envolvente que vamos a emplear. Si omitimos este parámetro, LOGO toma implícitamente una duración de 2 segundos, a no ser que se haya especificado una duración mayor.

La envolvente de tono puede tomar también un valor de 0 a 15, que representa el número de la envolvente que vamos a emplear. Si se omite este parámetro el ordenador produce un sonido de tono constante.

El último parámetro representa un valor del nivel de ruido del sonido y toma valor entre 0 y 31. Cuanto mayor sea este valor, menor es la frecuencia del ruido. Vamos a probar varios valores de este parámetro:

```
?sound [1 0 200 15 0 0 15] sound [1 0 200 15 0
0 31] sound [1 0 200 15 0 0 1]
?
```

Sólo puede emplearse los niveles de ruido de uno en uno; esto quiere decir que cualquier modificación de este parámetro anula el valor anterior.

7.2 env

Esta primitiva define el perfil de la envolvente de volumen que empleamos con la orden **sound**. Como ya se ha indicado, es posible definir hasta 15 envolventes distintas.

El nombre ''envolvente de volumen'' proviene del perfil que se obtiene al dibujar una gráfica de volumen/tiempo en unos ejes coordenados; el Manual del Usuario presenta varios ejemplos. Para definir la envolvente empleamos un primer parámetro, que es el número que la identifica, y a continuación hasta cinco grupos de tres parámetros cada uno, con un mínimo de un grupo de tres parámetros.

A continuación vamos a ver una instrucción **env** sencilla, seguida de una orden **sound** que hace uso de la misma:

```
?env [10 15 1 10]
?sound [2 300 200 0 10]
?
```

Esta envolvente tiene el número 10 y hace que el volumen aumente 15 escalones de una unidad cada uno, siendo la duración de cada escalón 10 centésimas de segundo. La orden **sound** produce un sonido en el canal B cuyo volumen aumenta desde cero hasta llegar a un máximo, terminando en este punto.

Los límites para los parámetros de **env** son: el número de escalones puede variar entre 0 y 127; la altura de cada escalón entre −128 y +127 (más adelante volveremos sobre este tema), y la duración de cada escalón puede ser de 1 a 256 centésimas de segundo.

Para finalizar con volumen cero puede añadir otro grupo de tres parámetros a la envolvente:

```
?env [10 15 1 10 5 253 40]
?sound [2 400 1000 0 10]
?sound [2 400 100 0 10]
?
```

con lo que el sonido aumenta suavemente, disminuyendo a continuación de forma bastante menos suave. El primer sonido se solapa con el segundo, dado que la envolvente dura más de 100 unidades de tiempo.

Es importante destacar el valor 253 que hemos dado a la altura del escalón; LOGO considera este valor equivalente a −3, ya que 253−256 es −3. El segundo grupo de parámetros establece otros 5 escalones con alturas de −3, es decir, disminuyendo, cada uno de los cuales dura 40 centésimas de segundo. Si se modifica ligeramente la envolvente de forma que los escalones tengan una altura equivalente a −2:

```
?env [10 15 1 10 5 254 40]
?sound [2 100 1000 0 10]
?
```

el sonido mantiene el nivel de volumen en 5 durante un buen rato. La disminución de volumen es un poco brusca, así que vamos a aumentar el número de escalones a 10, lo que nos permite darles menos altura (−1):

```
?env [10 15 1 10 255 40]
?sound [2 50 1000 0 10]
?
```

Ya hemos dicho que las envolventes pueden contener hasta cinco grupos de parámetros, por lo que disponemos de una gran libertad a la hora de definir las envolventes.

La diferencia principal a la hora de usar envolventes de volumen en BASIC y en LOGO reside en que éste último no admite valores negativos para fijar la altura de cada escalón. Si queremos definir escalones descendentes tenemos que hacerlo con el método de los complementos, es decir, restando el valor a 256. Es posible simplificar un poco estos cálculos de la siguiente manera:

```
?to _ :val
>op 256 - :val
>end
_ defined
?env (se 10 15 1 10 5 _ 2 40)
?
```

Se emplea el guión de subrayado (_) por su parecido con el signo menos, con lo que resulta bastante fácil de comprender y recordar.

7.3 ent

Esta es la primitiva que se usa para definir la envolvente de tono que empleamos en la instrucción **sound**. Es muy parecida a la envolvente de volumen **env**, aunque difiere de ésta en el número de escalones.

Vamos a ver un ejemplo sencillo:

```
?ent [15 239 1 5]
?sound [2 200 2000 12 0 15]
?
```

En este ejemplo, el tono va descendiendo gradualmente al crecer el parámetro tono hasta llegar a un nivel, en el que permanece hasta que finaliza. El valor original del parámetro tono es 200, tal y como podemos ver en la instrucción **sound**. Este parámetro aumenta en los 239 escalones de la envolvente, creciendo una unidad en cada escalón. La duración de cada uno de estos últimos es de 5 centésimas de segundo.

Se puede obtener un efecto totalmente distinto con la misma envolvente sin más que cambiar el valor inicial del tono:

```
?sound [2 20 2000 12 0 15]
?
```

La estructura es la misma, únicamente varían los valores del parámetro tono.

Los parámetros de la envolvente pueden tomar valores comprendidos entre los siguientes límites: el número de escalones varía entre 0 y 239; la altura de cada escalón entre −128 y +127, y, por último, la duración de cada escalón entre 1 y 256 centésimas de segundo.

Para conseguir una envolvente en que el tono del sonido baje para volver a aumentar a continuación y que dure exactamente lo mismo que la sentencia **sound** anterior, puede usar la siguiente secuencia:

```
?ent [15 239 2 5 161 254 5]
?
```

Fíjese que 254 es equivalente a 254−256, es decir, a −2. La duración de esta envolvente es de 239*5 + 161*5 = 2000 centésimas de segundo.

7.4 release

Cuando una instrucción **sound** produce el bloqueo de un canal de sonido, puede liberarlo por medio de la primitiva **release**. Esta primitiva identifica los canales con los mismos números que emplea **sound**, es decir, 1 para A, 2 para B y 4 para C.

Capítulo 8

Las potencias de 2

Este capítulo contiene dos aplicaciones que guardan estrecha relación con las potencias de 2. Ambas son ejercicios binarios. He elegido la primera de ellas por tratarse de un ejemplo muy frecuente que emplean muchos programadores en todos los lenguajes. La segunda guarda estrecha relación con la primera y se trata de una aplicación muy agradable.

Tanto con el primer ejemplo, que se trata de un juego, como con el segundo, que es un pequeño programa que dibuja árboles, podremos estudiar como controlar la recurrencia en LOGO. Aquellas personas que deseen estudiar la recurrencia desde otros puntos de vista, pueden consultar mi libro "Pocket Guide to LOGO", editado por Pitman Books.

8.1 El juego ALTOBAJO

Vamos a trabajar a continuación con un juego muy conocido que se emplea con frecuencia para enseñar técnicas de programación. Se trata del juego ALTOBAJO. En él, el ordenador elige un número **secreto** entre 0 y 99 y el jugador intenta adivinarlo. Si acertamos, el ordenador nos dice cuántos **intentos** hemos realizado; pero si fallamos nos dice si nuestro **numero** es más alto o más bajo que el número **secreto**.

Vamos a dar al procedimiento el mismo nombre que al juego, es decir, **altobajo**.

```
?to altobajo
>(local "secreto "numero "intento)
>make "intentos 0
>empezar
>jugar
>end
altobajo defined
?
```

Esta definición es un buen ejemplo de una técnica de programación que ya está probada y aceptada por muchos programadores; en caso de duda dele un nombre. No tengo la menor idea de cómo debe ser el programa del juego, al menos con detalle, pero lo que si sé con seguridad es que necesito por lo menos tres elementos, **secreto**,

numero e **intentos**, y que tengo que empezar haciendo que el ordenador elija su número **secreto** para poder empezar a **jugar**.

Según lo especificado arriba, el procedimiento **empezar** debe ser más o menos así:

```
?to empezar
>make "secreto random 100
>end
empezar defined
?
```

El procedimiento **jugar** no va a resultar tan sencillo. En primer lugar tenemos que incrementar el número de **intentos**; a continuación tomamos el **numero** que supone el jugador. En caso de que haya acertado no tenemos más que mostrar en pantalla el número total de **intentos**, pero si no ha acertado tenemos que decirle si su **numero** es demasiado alto o demasiado bajo, para volver a **jugar** a continuación.

```
?to jugar
>make "intentos :intentos + 1
>make "numero first rl
>if :numero = :secreto [(pr [acierto en]
:intentos) stop]
>if :numero < :secreto [pr [bajo]]
[pr [alto]]
>jugar
>end
jugar defined
?
```

Antes de intentar explicar cómo funciona el procedimiento, aquí tiene un ejemplo:

```
?altobajo
50
bajo
75
bajo
87
alto
81
bajo
84
alto
83
alto
82
acierto en 7
?
```

Ya ha visto cómo responde el ordenador. Este procedimiento es un ejemplo de recurrencia "controlada". En la recurrencia de cola de la que he hablado anteriormente no existe un verdadero control, pero en este juego si que lo hay. El procedimiento **jugar** incluye las variables **intentos, numero** y **secreto**, que son locales respecto a este procedimiento y globales respecto a los procedimientos contenidos en **altobajo, jugar** y **empezar**. La primera línea de **jugar** se encarga de incrementar el contador de **intentos** en 1; a continuación damos a **numero** el valor del primer **(first)** elemento de la lista que el jugador escribe por medio de la primitiva **rl**. Normalmente este elemento será un número.

Si el **numero** coincide con el valor **secreto**, se imprime en pantalla que el jugador ha acertado el valor y el número de intentos que ha necesitado; a continuación finaliza el procedimiento con **stop**. En caso de que el número sea inferior al valor **secreto**, aparece en pantalla la palabra **bajo** y se puede volver a **jugar**, al igual que sucede si el **numero** es demasiado **alto**.

La forma más rápida para descubrir un número consiste en emplear el "corte binario", es decir, en dividir el intervalo en que está contenido **secreto** en dos mitades iguales. Con este método nunca precisamos más de siete intentos (averigüe por qué usted mismo). Puede estudiar la estrategia del corte binario en el ejemplo anterior.

8.2 El árbol

El resto del capítulo trata del dibujo de árboles binarios y se basa en buena parte en mi libro "Introducing LOGO", editado por Granada Collins.

En primer lugar estudie las figuras 8.1 a 8.4, que evidentemente siguen una cierta progresión. Al dibujarlas he empleado tinta "blanca brillante" para que los dibujos queden más claros. El procedimiento empleado es el siguiente:

```
?to arbol :l :o
>if :o = 0 [stop]
>lt 45 fd :l
>arbol :l/2 :o - 1
>bk :l rt 90 fd :l
>arbol :l/2 :o - 1
>bk :l lt 45
>end
arbol defined
?
```

En primer lugar observe en la figura 8.1, y suponga que la tortuga parte del pico de la "uve".

Originalmente la tortuga apunta hacia arriba; a continuación gira 45 grados a la izquierda y luego avanza una longitud determinada (pongamos **:l**); posteriormente retrocede la misma longitud, gira a la derecha 90 grados, avanza la misma distancia

que antes y por último retrocede la misma cantidad y gira 45 grados a la izquierda. Con esto debe haber vuelto a la posición y orientación de partida.

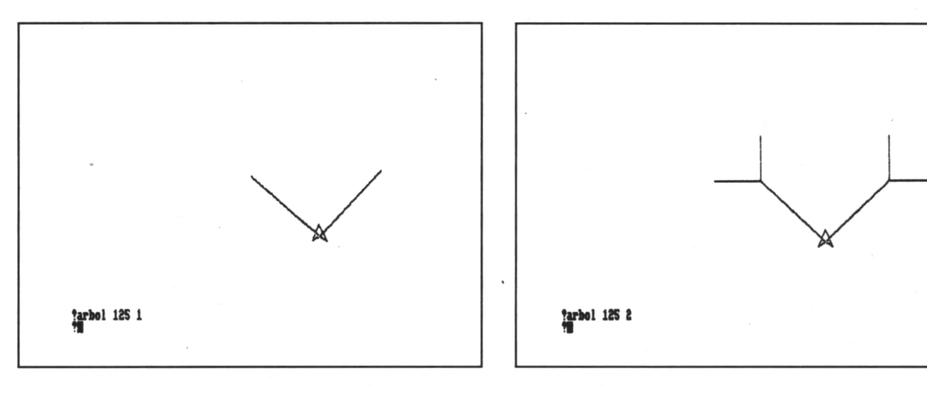


Fig. 8.1

Fig. 8.2

Fig. 8.3

Fig. 8.4

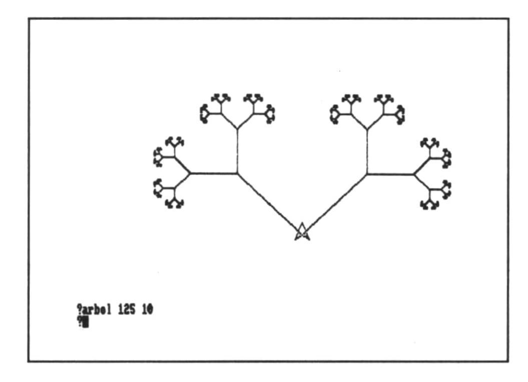


Fig. 8.5

La figura 8.2 procede de la 8.1, sólo que al final de cada avance la tortuga dibuja otra uve de mitad de tamaño que la precedente. Esta fórmula se va repitiendo en las figuras posteriores, pero la estructura básica es la de la primera figura, que es la que denominaremos de orden 1. La siguiente figura es la de orden 2, la siguiente de orden 3 y así sucesivamente.

A medida que crece el dibujo va comprendiéndose por qué se ha dado el nombre de "árbol" al procedimiento. Es muy fácil escribir el procedimiento **arbol**; no hay más que traducir a instrucciones de LOGO las palabras empleadas para describir el procedimiento. Es necesario añadir el concepto básico de "orden" de cada dibujo; este dato es la variable **:o** que aparece en el procedimiento, con lo que si el orden es 0 el procedimiento finaliza con **stop**. Cada vez que llamemos al procedimiento **arbol**, obtendremos una "uve" cuyo tamaño se reduce a la mitad de la anterior y cuyo orden se reduce en una unidad.

arbol es un procedimiento con recurrencia múltiple, dado que dentro de su propia definición se llama a sí mismo varias veces, aunque con distintos parámetros.

Para dibujar el primer "árbol" no hay más que llamar a **arbol :l 1**; de la misma forma, para dibujar el cuarto "árbol" se llama a **arbol :l 4**. La figura 8.5 se ha obtenido con **arbol :l 10**. Al llamar a este procedimiento con órdenes mayores, se obtendrán figuras que no se diferencian en la práctica de la anterior, dado que la resolución de la pantalla es limitada y no permite apreciar más detalles. Además, si la longitud inicial era, por ejemplo, 100 unidades, la décima llamada al procedimiento **arbol** tendrá una longitud de avance muy pequeña.

Para ver lo deprisa que disminuyen los términos de una progresión geométrica de razón 1/2, en la que cada término es la mitad del anterior, se puede aplicar el procedimiento **mitad**, que también tiene algo que ver con el corte binario.

```
?to mitad :l :o
>if :o = 15 [stop]
>(pr :o :l)
>mitad :l/2 :o + 1
>end
mitad defined
? mitad 100 0
0 100
1 50
2 25
3 12.5
4 6.25
5 3.125
6 1.5625
7 0.7812
8 0.390625
9 0.1953125
10 9.765625e-02
```

```
11 4.8828125e-02
12 2.44140625e-02
13 1.220703125e-02
14 6.10351562e-03
?
```

Vamos a volver un momento al juego ALTOBAJO. Si se fija en la tabla anterior en el valor que corresponde al orden 7, veremos que es inferior a la unidad. Esta es la razón que permite asegurar que, empleando el corte binario, nunca necesitamos más de siete intentos para adivinar un número comprendido entre 0 y 99, dado que la diferencia entre números consecutivos es de una unidad.

8.3 Un árbol inclinado

El árbol binario dibujado es un poco soso; sería mucho más interesante intentar dibujar árboles inclinados. La idea básica para dibujar estos árboles consiste en que una de las ramas que forman la "uve" crezca más que la otra. Con este sistema es posible conseguir unos efectos estupendos.

La figura 8.6 ha sido creada con la siguiente instrucción:

```
?cs pu fs bk 150 pd arbol_inc 150 20 1 .6 .5
?
```

La cosa está muy clara: alzamos la pluma (para que no pinte), pasamos a una pantalla de gráficos completa, retrocedemos 150 unidades, bajamos la pluma y ejecutamos **arbol_inc** con los cinco parámetros indicados.

Las dos figuras siguientes, 8.7 y 8.8, se obtienen con una instrucción similar, modificando ligeramente los parámetros de **arbol_inc**. La primera se consigue con **arbol_inc 150 20 2 .6 .5** y la segunda con **arbol_inc 150 20 3 .6 .5**. La figura 8.9 procede de **arbol_inc 150 20 10 .6 .5**, y demuestra cómo pueden obtenerse dibujos realmente agradables a partir de unos pocos elementos muy simples. El ordenador tarda un poco más en realizar este último dibujo, dado que lleva bastante más trabajo.

La definición del procedimiento **arbol_inc** es la siguiente:

```
?to arbol_inc :l :a :o :f1 :f2
>if :o = 0 [stop]
>lt :a
>fd :l
>arbol_inc (:l * :f1) :a (:o - 1) :f1 :f2
>bk :l
>rt (2 * :a)
>fd (:l * :f2)
```

```
>arbol_inc (:l * :f1 * :f2) :a (:o - 1) :f1 :f2
>bk (:l * :f2)
>lt :a
>end
arbol_inc defined
?
```

Realmente se trata de un procedimiento muy interesante y merece la pena dedicarle un poco de tiempo.

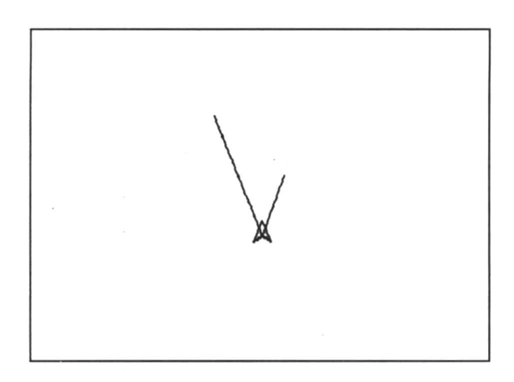


Fig. 8.6

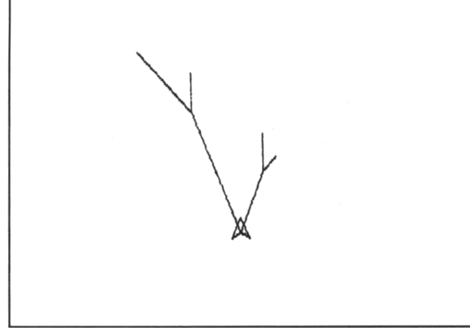


Fig. 8.7

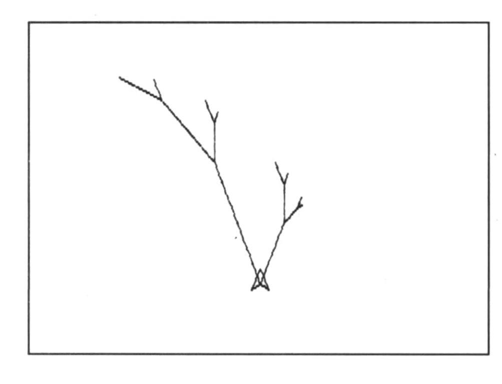


Fig. 8.8

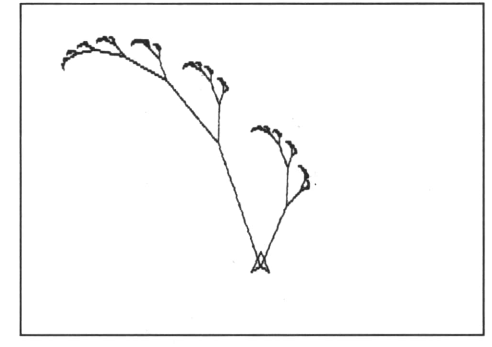


Fig. 8.9

Para estudiarlo a fondo, a continuación tiene un procedimiento similar en el que no están incluidas las órdenes a la tortuga:

```
?to arbol_int :l :a :o :f1 :f2
>if :o = 0 [stop]
>(pr :l :a :o :f1 :f2)
>arbol_int (:l * :f1) :a (:o - 1) :f1 :f2
```

```
>arbol_int (:l * :f1 * :f2) :a (:o - 1) :f1 :f2
>end
arbol_int defined
?arbol_int 1000 40 4 .8 .5
1000 40 4 0.8 0.5
800 40 3 0.8 0.5
640 40 2 0.8 0.5
512 40 1 0.8 0.5
256 40 1 0.8 0.5
320 40 2 0.8 0.5
256 40 1 0.8 0.5
128 40 1 0.8 0.5
400 40 3 0.8 0.5
320 40 2 0.8 0.5
256 40 1 0.8 0.5
128 40 1 0.8 0.5
160 40 2 0.8 0.5
128 40 1 0.8 0.5
64 40 1 0.8 0.5
?
```

Diviértase un poco estudiando el ejemplo anterior.

Capítulo 9

Nombres y propiedades

Vamos a dedicar una parte de este capítulo al estudio del generador de números aleatorios de Dr LOGO. Aprovechando este estudio analizaremos una de las características más importantes de LOGO: las "propiedades".

Los ejemplos de programación que vamos a ver son muy útiles, dado que pueden ayudarnos a distinguir perfectamente y de una vez por todas el nombre de un objeto de sus diversas propiedades. El contenido de este capítulo no es demasiado sencillo, pero hay que dominar perfectamente el concepto de propiedades para estar en condiciones de realizar programas de una cierta dificultad.

9.1 Los valores de las propiedades

Antes de seguir adelante, reinicialice LOGO desde BASIC.

Si escribe:

```
?.contents
```

LOGO muestra una lista de los contenidos de su área de trabajo. Ahora escriba:

```
?glist ".APV
[ ]
?
```

que da el resultado de aplicar la instrucción **glist** al parámetro **".APV**, es decir, una lista vacía. Observe que las comillas son muy importantes, ya que aclaran que nos referimos al nombre **.APV**. De momento vamos a olvidarnos de **glist** para asignar un valor con la instrucción **make**:

```
?make "a "bb
?:a
bb
?glist ".APV
[a]
?
```

En primer lugar el objeto denominado **a** toma como valor la palabra **bb**; LOGO sabe que nos referimos al nombre y a la palabra gracias a las comillas. A continuación pedimos a LOGO el valor del objeto introduciendo **:a**; la respuesta es **bb**, lógicamente. Finalmente solicitamos **glist ".APV**, a lo que el sistema responde con **[a]**; esto quiere decir que el elemento **APV** es realmente una "propiedad" que corresponde en concreto al valor de una variable.

La instrucción **glist** pregunta al sistema si existen objetos con la propiedad **.APV**. Cuando realizamos esta pregunta por primera vez, no había ningún objeto con valor y por eso obtuvimos una lista vacía. Ahora el valor de **a** es **bb**; por tanto si escribe:

```
?plist "a
[.APV bb]
?
```

recibirá una lista con las propiedades de **a**. Pruebe con:

```
?a
I don't know how to a
? to a
>pr [el procedimiento llamado a]
>end
a defined
?plist "a
[.DEF [ ]] [pr [el procedimiento llamado a]]]
.APV bb]
?
```

que indica en la primera línea del listado que LOGO considera **a** como el nombre de un procedimiento y, como no está definido previamente, no lo reconoce. En las líneas posteriores hacemos que **a**, además de tener un valor (**.APV**), tenga una definición como procedimiento (**.DEF**).

El procedimiento **a** produce una acción:

```
?a
el procedimiento llamado a
?
```

que usted preveía perfectamente a partir de la definición del procedimiento. A esta última le corresponde una propiedad, **.DEF**, cuyo valor es la propia definición, que no es más que una lista con otras listas dentro. Para aclarar un poco la forma en que LOGO guarda la definición en forma de lista, modifique la definición del procedimento **a**:

```
?to a :nada
>pr :nada
>end
a defined
?plist "a
[.DEF [[nada] [pr :nada]] .APV bb]
?
```

La mejor forma de estudiar cómo almacena LOGO las definiciones de los procedimientos es a través de la orden **grop**, como por ejemplo:

```
?gprop "a ".DEF
[[nada] [pr :nada]]
?
```

La orden **grop** puede emplearse para imitar a otra orden que aparece en muchas otras versiones de LOGO: la orden **thing**. Vamos a definir un procedimiento cuya respuesta sea el valor correspondiente a la propiedad **.APV**.

```
?to thing :nombre
>op gprop :nombre ".APV
>end
thing defined
?plist "thing
[.DEF [[nombre] [op gprop :nombre ".APV]]]
?thing "a
bb
?
```

Con las propiedades puede crearse muchas otras primitivas con las que cuentan otras versiones de LOGO, pero antes de seguir adelante escriba:

```
?ed "a
```

con lo que comprobará que el editor ofrece tanto el valor de **a** como su definición. Deberá obtener la siguiente respuesta:

```
to a :nada
pr :nada
end
make "a "bb
```

dado que el editor comprueba tanto si el nombre tiene propiedad de valor como si tiene propiedad de definición. La versión para Amstrad de Dr LOGO tiene aún otra propiedad estándar: la propiedad **.PRM**. Esta propiedad da la posición de memoria en la que reside la primitiva de que se trate.

Para obtener una lista de todas las primitivas del sistema, es decir, para obtener el equivalente a **.contents** pero sólo con primitivas, se puede definir el siguiente procedimiento:

```
?to .primitivas
>op glist ".PRM
>end
.primitivas defined
?
```

De la misma forma, para obtener una lista de los nombres de variables se puede emplear:

```
?to .nombres
>op glist ".APV
>end
.nombres defined
?
```

Más adelante podría ser muy interesante definir un procedimiento que además de los nombres de las variables proporcionase sus valores.

9.2 Los valores de los valores

Suponga que tenemos definido el procedimiento **thing**. Haga el siguiente experimento:

```
?make "a "b
?make "b "c
?thing "a
b
?:a
b
?thing "b
c
?thing thing "a
c
?thing :a
c
?::a
:a has no value
?
```

Se ha dado al objeto llamado **a** el valor **b**, y al objeto llamado **b** el valor **c**.

thing "a da el valor **b,** al igual que **:a.** De la misma forma, **thing "b** da el mismo valor que **:b,** es decir **c.** Puede comprobarlo tanto con **thing thing a** como con **thing :a.** De todo esto se deduce que los dos puntos (:) son equivalentes a **thing,** pero esto no quiere decir que podamos intercambiar el uno por el otro en todos los casos. El ordenador admite **thing thing a,** pero por el contrario no admite **::a,** puesto que LOGO no reconoce **:a** como el nombre de un objeto.

Al escribir por primera vez la letra **a,** el ordenador ha respondido "**I don't know how to a**". Pruebe qué sucede al usar números:

```
?1
1
?plist 1
plist doesn't like 1 as input
?plist "1
[ ]
?make 1 2
make doesn't like 1 as input
?make "1 2
?plist "1
[.APV 2]
?thing "1
2
?:1
2
?
```

Se ha obtenido un objeto, denominado **1,** cuyo valor como propiedad **.APV** es **2.** Literalmente, "el valor de **1** es **2**" (?!).

Antes de pasar a los procedimientos con números aleatorios, realice un último experimento:

```
?plist "to
[ ]
?to to
>pr [que rara]
>end
to defined
?to
isn't a name or procedure
?plist "to
[.DEF [[ ] [que rara]]]
?
```

El único cambio ha sido dar a la primitiva **to** de LOGO una definición nueva; sin embargo LOGO no tiene en consideración esta nueva definición de **to,** y sigue traba-

jando con ella como antes. Si investiga la propiedades de **to** encontrará que **.DEF** figura en la lista.

LOGO tiene ciertos criterios de prioridad que le permiten decidir qué hacer cuando se encuentra con propiedades contradictorias. Uno de estos criterios indica que las primitivas tienen prioridad sobre las definiciones posteriores que pueda realizar el usuario.

9.3 Pruebas aleatorias

En LOGO una sucesión de caracteres forma una palabra. Para formar una palabra con varios caracteres consecutivos, Dr LOGO cuenta con la primitiva **word**. Escriba las siguientes líneas teniendo cuidado de no confundirse con los paréntesis:

```
?word "xx "999
xx999
?(word "xx "999 "yy)
xx999yy
?(word "xx)
xx
?word "xx
Not enough inputs to word
?
```

Fíjese cómo se emplean los paréntesis para agrupar las primitivas que pueden tener un número variable de parámetros. **word** es una de estas primitivas; habitualmente emplea dos parámetros, pero admite un número superior o inferior. Siempre que los parámetros no sean exactamente dos, es necesario encerrar toda la sentencia entre paréntesis, incluyendo la propia orden.

El nombre **"1** no es lo mismo que el número **1**; incluso el nombre admite un valor diferente a **1**. Para dejar las cosas claras se puede emplear la orden **word**. Supongamos que tenemos un objeto denominado **"9** y que queremos sumar una unidad a su contenido (**.APV**). Para ello podemos emplear varios métodos:

```
?make "9 10
?make "9 :9 + 1
?make "9 (thing "9) + 1
?make (word 9) (thing (word 9)) + 1
?
```

La instrucción **thing** no trabaja con un número de parámetros variable; no obstante, está encerrada entre paréntesis porque de lo contrario el ordenador interpreta **thing ("9+1)** como **thing "9 + 1**, o sea, **thing "10**. Haga la prueba:

```
?thing "9 + 1
 gprop doesn't like 10 as input in thing:
 op gprop :nombre ".APV
?
```

LOGO considera **10** como un número y **"10** como una sucesión de caracteres que constituyen un nombre. Otra forma de realizar la misma tarea consiste en colocar el operador aditivo entre el número y la instrucción de LOGO.

A continuación vamos a volver a emplear el procedimiento **randomize** que hemos visto en un capítulo anterior y cuya definición aparece un poco más abajo. Junto a él vamos a ver otros procedimientos que nos van a servir para comprobar la disposición de los números aleatorios.

```
?to pops
>po glist ".DEF
>end
pops defined
?
```

Hemos incluido en primer lugar este procedimiento porque su objetivo es proporcionar una lista de todos los procedimientos existentes. Algunas versiones de LOGO cuentan originalmente con la primitiva **posp**, pero en nuestro caso no es así y por tanto tenemos que crearla de esta manera. La orden **glist ".DEF** nos da una lista con todos los objetos que tienen la propiedad, es decir, de todos los procedimientos. Las definiciones de éstos pueden sacarse en pantalla mediante la instrucción **po**, cuyo parámetro es la mencionada lista.

Otra instrucción frecuente en otras versiones de LOGO es **edall**, que se encarga de editar todos los procedimientos que hay en la memoria del ordenador. Evidentemente, es posible crear esta instrucción con el siguiente procedimiento:

```
?to edall
<ed glist ".DEF
<end
edall defined
?
```

Pero volvamos a los números aleatorios.

Introduzca los siguientes procedimientos para comprobar si existe alguna ley que ligue los números aleatorios:

```
?to comprobar :num :rep
>borrar :num - 1
>repeat :rep [mas :num]
>listar :num - 1
>end
comprobar defined
?to mas :x
>local "n
>make "n random :x
```

```
>make (word :n) 1 + thing (word :n)
>end
mas defined
?to randomini
>repeat nodes - 50 * int (nodes / 50) [if = 2
random 2 [ ]]
>end
randomini defined
?to borrar :n
>make (word :n) 0
>if not = :n 0 [borrar :n - 1]
>end
borrar defined
?to listar :n
>(pr :n thing (word :n))
>if not = :n 0 [listar :n - 1]
>end
listar defined
?
```

El sistema empieza con el procedimiento **comprobar:**, que emplea dos parámetros: **:num**, que fija el intervalo de números a comprobar, y **:rep**, que representa el número de veces que vamos a llamar a la primitiva **random**. El primer paso es dar el valor 0 a los objetos cuyos nombres están comprendidos entre **0** y **:num - 1**. El segundo es ejecutar **:rep** veces el procedimiento **mas :num**, para activar finalmente **listar :num - 1**.

El procedimiento **mas :x** comienza por declarar una variable local, **:n**; de esta forma el campo de acción de **n** queda restringido a este procedimiento. A continuación se da a **n** un valor comprendido en el intervalo **0** a **:num - 1**, como consecuencia de la orden **random** y del parámetro **:num**. Posteriormente el contenido de **:n** es convertirlo en nombre por medio de la primitiva **word** y, finalmente, aumenta en una unidad el contenido de dicho nombre por medio de **1 + thing (word :n)**.

De todo lo anterior se deduce que esta rutina se encarga de tomar un número aleatorio, convertirlo en el nombre de una variable y finalmente incrementar su contenido en una unidad.

El procedimiento que se encarga de poner a cero los valores de los que antes habíamos hablado es **borrar :n**. Comienza poniendo a cero la variable cuyo nombre corresponde al número que contiene **:n** y, en caso de que **:n** no sea igual a cero, el procedimiento vuelve a llamarse a sí mismo, disminuyendo en una unidad el valor de su parámetro. Es un ejemplo clásico de utilización de las técnicas de recurrencia.

Si nos detenemos a comparar los procedimientos **listar** y **borrar**, veremos que ambos tienen estructuras similares. Sólo se diferencian en la primera línea, dado que el primer procedimiento se encarga de sacar en pantalla los valores de los objetos cuyos nombres están comprendidos en el intervalo **:n - 1** a **0**.

En pocas palabras, estos procedimientos se encargan de generar **:rep** números aleatorios comprendidos entre **0** y **:num − 1**, sacando en pantalla una lista de sus valores. A continuación tiene unos cuantos ejemplos:

```
?randomini comprobar 10 1000
9 111
8 93
7 94
6 96
5 95
4 105
3 104
2 109
1 99
0 94
?comprobar 10 1000
9 104
8 109
7 93
6 116
5 98
4 94
3 85
2 100
1 96
0 105
?comprobar 10 1000
9 107
8 96
7 108
6 90
5 109
4 112
3 94
2 90
1 97
0 97
?comprobar 10 5000
9
488
8 486
7 490
6 494
5 505
4 500
```

3 527
2 508
1 510
0 492
?

Los resultados obtenidos parecen confirmar que el generador de números aleatorios que emplea el LOGO de Amstrad es realmente aleatorio.

El ordenador necesita bastante tiempo para ejecutar este procedimiento, pero no olvide que si introducimos, por ejemplo, **comprobar 10 5000**, estamos llamando 50000 veces a la instrucción **random**.

En el próximo capítulo volveremos a emplear nuestros conocimientos sobre las propiedades para estudiar la aritmética en LOGO.

Capítulo 10

Aritmética y estadística

Cuando empezamos a jugar con los números en LOGO, se abre ante nuestros ojos un mundo totalmente nuevo. Si introduce las siguientes líneas, obtendrá resultados bastante interesantes.

```
?make "x 123
?plist "x
[.APV 123]
?:x
123
?make "x 1/3
?plist "x
[.APV 0.333333333333333]
?:x
0.333333333333333
?count 123
3
?count 1/3
17
?
```

Antes de seguir adelante trate de averiguar qué es lo que ha sucedido.

10.1 Palabras y números

En primer lugar hemos dado a la variable **x** el valor **123**, lo que podemos comprobar solicitando una lista de las propiedades de **x** o bien por medio de **:x**. En segundo lugar hemos dado a **x** el valor **1/3**, equivalente a **0.333333333333333**, y finalmente hemos averiguado el número de elementos que contienen varias palabras por medio de la instrucción **count**.

Observe que en **123** hay tres elementos, mientras que en **1/3** hay **17** elementos. ¿Por qué? Vuelva a los contenidos de **x**.

Cuando Dr LOGO encuentra un número entero, como es **123**, puede almacenarlo de forma exacta en una palabra de tres elementos; pero cuando tropieza con algo que debería ser un número y que no puede almacenar de forma exacta con todos sus decimales, lo convierte en un número lo mejor que puede. LOGO no puede almacenar **1/3** como número con este formato, así que realiza la división y guarda el resultado con quince cifras decimales significativas.

Las quince cifras decimales significativas más el punto decimal y el primer cero nos dan un total de 17 elementos. Se podría haber llegado al mismo resultado de otra forma distinta:

```
?item 1 1/3
0
?item 2 1/3
.
?
```

Según estas instrucciones, el primer elemento (**item 1**) de la lista de caracteres que constituye el número **1/3** es un cero, mientras que el segundo es un punto. Aún quedan unos cuantos detalles que afectan a los operadores aritméticos.

Haga unas cuantas pruebas con lo siguiente:

```
?1234567890987654321
1.23456789098765e+18
?"1234567890987654321
1234567890987654321
?"1234567890987654321/1
1.23456789098765e+18
?
```

Compruebe que LOGO transforma el número **123456789098764321** en notación científica, es decir, **123456789098765e+18**. El ordenador trata la palabra **"1234567890987654321** como tal, y no como un número, a causa de las comillas que la preceden. Sin embargo, la expresión **"1234567890987654321/1** recibe un tratamiento diferente. LOGO trata la primera parte como una palabra, al igual que antes, pero al encontrarse el operador de división, **/**, convierte la palabra en número, lo divide por **1** y finalmente da el resultado como número.

Suponga que queremos que el ordenador trate la palabra **1/3** como tal. ¿Qué tenemos que hacer? Es preciso informar a LOGO de que no queremos que trate el signo / como operador de división, sino que lo considere como un caracter más. Para ello podemos emplear CTRL Q delante del carácter en cuestión, o bien pulsar directamente en el teclado la barra inclinada hacia la izquierda. Observe que el resultado de CTRL Q es precisamente incluir esa barra. Observe qué diferencias hay entre los siguientes ejemplos:

```
?"1/3
0.333333333333333
?"1\/3
1/3
?\1\/\3
I don't know how to 1/3
?
```

10.2 Prioridades en aritmética

En lo que resta de capítulo, he intentado crear un sistema que permita realizar cálculos estadísticos sencillos de una o dos variables con dos grupos de números.

La operación más frecuente en estadística es la suma. Para aquéllos que estén interesados en el tema aconsejo mi libro "Pocket Guide to Statistical Programming", editado por Pitman Books. Ya sabemos que la forma básica de almacenar información en LOGO es la lista y, por tanto, los cálculos estadísticos en LOGO requieren su empleo en operaciones y el manejo de las que contengan la información que nos interese.

Para sumar los elementos de una lista no hay más que añadir el valor del primer elemento a la suma de los restantes, que a su vez se obtiene sumando el valor del primero a la suma de todos los demás. Es evidente que se trata de otro ejemplo clásico de procedimientos recurrentes.

A continuación está definido un procedimiento, al que llamaremos **sigma** para empezar a emplear la terminología estadística, que se encarga de calcular la suma de los elementos de una lista empleando un método recurrente.

```
?to sigma :listadatos
>if not (count :listadatos) = 1 [op
(first :listadatos + sigma bf :listadatos)]
>if (count :listadatos) = 1 [op first
:listadatos]
>end
sigma defined
?
```

En caso de que el número de elementos de la lista no sea igual a 1, **sigma** imprime en la pantalla el resultado de sumar el primer elemento de la lista a lo que hemos obtenido aplicando el mismo procedimiento al resto de los elementos. Por el contrario, en caso de que la lista solamente contenga un elemento, el resultado será el valor contenido en el primer y único elemento de la misma.

Veamos un ejemplo:

```
?first [a b c d] bf [a b c d]
a
[a b c]
?sigma [1 2 3 4]
+ doesn't like [1 2 3 4] as input in sigma:
if not ( count :listadatos ) = 1 [op (first :
listadatos + sigma bf :listadatos)]
?
```

con lo que volvemos a encontrarnos con un problema que ya nos es familiar. El operador aditivo intenta sumar los elementos que tiene a ambos lados, pero en este caso nos encontramos con **:listadatos + sigma bf :listadatos**. LOGO intenta sumar **:listadatos** (que en este caso no es otra cosa que **[1 2 3 4]**) con el resultado de **sigma**. Es preciso indicar al ordenador el orden de prioridades a la hora de realizar las operaciones; para ello podemos emplear los paréntesis, o bien hacer uso de otra forma de emplear el operador aditivo.

Cuando el operador + está situado entre los dos datos a sumar decimos que se trata de un operador "infijo". La versión de LOGO que emplea Amstrad también admite el uso del operador aditivo delante de los datos, como si se tratase de una instrucción. Cuando lo empleamos de esta forma decimos que estamos trabajando con un operador "prefijo". Compruébelo con las siguientes instrucciones:

```
?1 + 3
4
?+ 1 3
4
?1 - 3
-2
?- 1 3
-2
?-1 3
-1
3
?* 1 3
3
?/ 1 3
0.333333333333333
?
```

Fíjese que el ordenador interpreta **−1 3** como dos números, un **−1** seguido por un **3**, mientras que **− 1 3** pasa a ser **−2**. El espacio que hay entre el signo menos (−) y el primer número es muy importante. También se puede emplear en forma de pre-

fijo el operador de igualdad (=). Con todo esto es posible realizar otra versión del procedimiento **sigma**:

```
?to sigma :listadatos
>if not = count :listadatos 1 [ op +
first :listadatos sigma bf :listadatos]
>if = count :listadatos 1 [op first :listadatos]
>end
sigma defined
?sigma [1 2 3 4]
10
?
```

El próximo paso es sumar las informaciones contenidas en dos listas distintas.

10.3 Combinación de listas

En primer lugar, fíjese en las siguientes sentencias de LOGO. Todas ellas emplean operadores prefijos.

```
?* 2 3
6
?[* 2 3]
[* 2 3]
?run [* 2 3]
6
?(se "* "2 "3)
[* 2 3]
?run (se "* "2 "3)
6
?make "oper "* make "p1 2 make "p2 3
?(se :oper :p1 :p2)
[* 2 3]
?run (se :oper :p1 :p2)
6
?
```

La primitiva **se**, abreviatura de sentencia, toma una serie de datos y construye una lista con ellos. Por su parte, la orden **run** se encarga de tomar una serie de instrucciones y ejecutarlas como si se tratase de parte de un programa.

Hemos tomado tres objetos (**:oper :p1 :p2**), el primero de los cuales representa un operador, y los hemos convertido en instrucciones que forman un corto programa.

Éste es un procedimiento que realiza esta misma acción:

```
?to runop :oper :p1 :p2
>op run (se :oper :p1 :p2)
>end
runop defined
?runop "* 2 3
6
?
```

pero lo que realmente queremos hacer es aplicar un operador a cada par de elementos que tomemos de dos listas, así que no tenemos más que aplicar **runop** y formar otra lista con **se**.

```
?to combinar :oper :l1 :l2
>if not = 1 count :l1 [op (se runop :oper
first :l1 first :l2 combinar :oper bf :l1 bf
:l2)]
>if = 1 count :l1 [op runop :oper first :l1
first :l2]
>end
combinar defined
?combinar "* [1 2 3 4 5] [1 2 3 4 5]
[1 4 9 16 25]
?
```

Una vez que hemos definido este procedimiento, para calcular la suma de los cuadrados de los elementos de la lista **[1 2 3 4 5]** no tenemos más que hacer:

```
?sigma combinar "*[1 2 3 4 5] [1 2 3 4 5]
55
?
```

Merece la pena que nos detengamos un momento para analizar con detalle el procedimiento **combinar**.

Para ello vamos a servirnos de un pequeño truco. En primer lugar introduzca **ed "combinar** y, una vez que se encuentre en el editor, cambie el nombre del procedimiento por **comb**, las llamadas a **combinar** que realiza el mismo procedimiento por llamadas a **comb**, y por último añada una línea.

```
?to comb :oper :l1 :l2
>pr list :l1 :l2
>if not = 1 count :l1 [op (se runop :oper first
:l1 first :l2 comb :oper bf :l1 bf :l2)]
>if = 1 count :l1 [op runop :oper first :l1
first :l2]
>end
```

La línea añadida puede ayudar bastante a la hora de seguir la pista a los datos de **comb**, o lo que es lo mismo, a los datos de **combinar**.

```
?comb "* [1 2 3] [1 2 3]
[1 2 3] [1 2 3]
[2 3] [2 3]
[3] [3]
[1 4 9]
?pr se [1 2 3] [1 2 3]
1 2 3 1 2 3
?pr list [1 2 3] [1 2 3]
[1 2 3] [1 2 3]
```

Ahora es posible ver que las listas introducidas como datos del procedimiento **combinar** tienen un elemento menos cada vez que llamamos al procedimiento. Las líneas anteriores ofrecen, asimismo, un ejemplo de utilización de la primitiva **list** y de las diferencias entre **list** y **se**. Ya sabemos que **se** une los elementos y forma una sola lista; por el contrario, **list** conserva las listas componentes como tales.

Se puede emplear la instrucción **list** para realizar las funciones de **lput**, una primitiva con la que cuentan otras versiones de LOGO. La instrucción **lput** es la opuesta a **fput** (**f** y **l** son iniciales de **first** y de **last**, en inglés **primero** y **último**)

```
?fput "x [xx yy [zz]]
[x xx yy [zz]]
?se "x [xx yy [zz]]
[x xx yy [zz]]
?fput [x] [xx yy [zz]]
[[xx] xx yy [zz]]
?se [x] [xx yy [zz]]
[x xx yy [zz]]
?list "x [xx yy [zz]]
[x [xx yy [zz]]]
?list [x] [xx yy [zz]]
[[x] [xx yy [zz]]]
?
```

De todo esto puede deducirse que debemos emplear **se** cuando el elemento que queremos añadir a una lista es una palabra, por ejemplo **"x**, reservando **list** para aquellos casos en los que lo que queramos añadir sea una lista a otra, por ejemplo **[x]**.

Ahora bien, si queremos imitar el funcionamiento de **lput** tenemos que distinguir dos casos, dado que no es lo mismo añadir una palabra a una lista que añadir otra lista. Para ello podemos emplear **wordp**:

```
?wordp "x
TRUE
?wordp 3
TRUE
?wordp [x]
FALSE
?to lput :elem :listadatos
>if wordp :elem [op se :listadatos :elem]
[op list :listadatos :elem]
>end
lput defined
?
```

Personalmente no estoy satisfecho con esta definición; me parece mucho más lógico que el orden de los datos sea **:listadatos :elem**, pero la norma general es el orden inverso que hemos empleado en nuestro caso.

10.4 Estadística con dos variables

Los elementos básicos con los que trabaja la estadística son las medias, las desviaciones típicas y las covarianzas.

La media es la suma de todos los elementos de una lista dividida por el número de elementos que contiene. El procedimiento **media** es muy sencillo de definir:

```
?to media :listadatos
>op / sigma :listadatos count :listadatos
>end
media defined
?media [1 2 3 4]
2.5
?
```

Otra forma de obtener el mismo resultado es **op (sigma :listadatos) / (count :listadatos)**, con el operador de división intercalado. Personalmente prefiero el primer sistema con operador prefijo; me parece más elegante. La media de **[1 2 3 4]** sería igualmente **(1 + 2 + 3 + 4) / 4 = 2.5**.

Otra magnitud muy útil es el valor de la media de los cuadrados de los elementos de una lista. Para ello hay que elevar al cuadrado cada elemento, sumar todos los cuadrados y finalmente dividir este resultado por el número de elementos de la lista.

El procedimiento para calcular este valor es el siguiente:

```
?to mediacuad :listadatos
>op media combinar "* :listadatos :listadatos
>end
mediacuad defined
?mediacuad [1 2 3 4] combinar "* [1 2 3 4]
[1 2 3 4] media [1 4 9 16]
7.5
[1 4 9 16]
7.5
?
```

Por último, no queda más que determinar otro valor para poder realizar todos los cálculos estadísticos con dos variables.

Esta magnitud es la media de los productos cruzados de los elementos correspondientes de las dos listas:

```
?to mediaprod :l1 :l2
>op media combinar "* :l1 :l2
>end
mediaprod defined
?mediaprod [-2 -1 0 1 2] [-2 -1 0 -1 -2]
0
?
```

El siguiente paso es emplear los procedimientos ya definidos para determinar la varianza de los valores de una lista y la covarianza de los elementos de un par de listas.

La varianza de una lista de valores es igual a la diferencia entre el valor de la media de sus cuadrados y el cuadrado de su media aritmética.

```
?to varianza :listadatos :mediadatos
>op - mediacuad :listadatos (* :mediadatos
:mediadatos)
>end
varianza defined
?varianza [1 2 3 4 5] (media [1 2 3 4 5])
2
?
```

El procedimiento **varianza** emplea dos parámetros: la lista de los datos y su media aritmética. En este ejemplo la instrucción encargada de determinar el cuadrado de la media va entre paréntesis. He empleado el segundo parámetro porque la media

se usa en muchos sitios, incluyendo el procedimiento para determinar la covarianza que vamos a ver a continuación. Podría haber llamado tranquilamente al procedimiento **media**, pero, si se calcula una sola vez este valor y se introduce en una variable, nos ahorraremos unos cuantos cálculos cuando haya que volver a utilizarlo.

De momento no se preocupe por este detalle; un poco más adelante veremos claramente su utilidad. Ahora veamos el procedimiento **covarianza**. La covarianza de dos listas de valores es igual a la media del producto cruzado de ambas menos el producto de las dos medias, es decir:

```
?to covar :l1 :l2 :m1 :m2
>op - mediaprod :l1 :l2 (* :m1 :m2)
>end
covar defined
?covar [1 2 3 4 5] [5 4 3 2 1] media
[1 2 3 4 5] media [1 2 3 4 5]
-2
?
```

El resto de los cálculos estadísticos de este capítulo se basan en los conceptos de varianza y covarianza que acabamos de definir, pero antes de seguir adelante debemos definir dos procedimientos para calcular dos funciones matemáticas muy útiles.

10.5 Un invento aritmético

La magnitud de uso más frecuente de una lista de valores, después de su media aritmética, es su desviación típica, que es igual a la raíz cuadrada de la varianza. Desgraciadamente, la versión de LOGO que emplea Amstrad no cuenta con una primitiva que permita calcular raíces cuadradas, así que tendremos que ingeniárnoslas para definir un procedimiento que realice esta tarea. La cosa es bastante sencilla.

Aquí tiene un procedimiento que calcula raíces cuadradas por el método de Newton:

```
?to raiz2 :num
>local "r2
>make "r2 :num / 2
>repeat 128 [if abs (:r2 - :num/:r2) < :r2 *
5.e-15 [op :r2] [make "r2 (:r2 + :num/:r2) / 2]]
>op :r2
>end
raiz2 defined
?to abs :a
>if :a < 0 [op -:a] [op :a]
>end
abs defined
?
```

Para ver cómo funciona **raiz2** añada la instrucción **pr :r2** delante de la sentencia **if**, es decir, **(pr :r2 if abs...)**. Los valores que aparecen en la pantalla convergen hacia un determinado valor que es la raíz cuadrada del número. Por ejemplo:

```
?raiz2 10
5
3.5
3.17857142857143
3.16231942215088
3.16227766044414
3.16227766016838
3.16227766016838
?
```

El procedimiento **raiz2** emplea un bucle sin fin que modifica continuamente el valor de la variable local **r2**. El ordenador calcula la diferencia en valor absoluto entre cada dos valores de **r2**, y sale del bucle cuando esta diferencia alcanza el orden de 5.e−15, imprimiendo en pantalla el valor de **r2**. Los valores de las dos últimas líneas son iguales, dado que el primero de ellos procede de la instrucción **pr** y el segundo de **op**.

Para comprobar que el procedimiento funciona correctamente, haga una prueba con lo siguiente:

```
?raiz2 10
3.16227766016838
?raiz2 10 "y "luego 10/raiz2 10
3.16227766016838
y
luego
3.16227766016838
?
```

Puede observar que el valor que hemos calculado con **raiz2 10** es igual al valor de **10/raiz2 10**, como tenía que ser.

El procedimiento **abs** es muy simple; se encarga de determinar el valor absoluto de un parámetro:

```
?abs 3 abs 0 abs −0 abs −4
3
0
0
4
?
```

Ahora ya estamos en condiciones de hacer frente al resto de los cálculos estadísticos.

10.6 Correlación y regresión

Una vez definidos todos los procedimientos anteriores, calcular la correlación de dos listas de valores es una tarea verdaderamente sencilla. La correlación es igual a la covarianza de las dos listas dividida por el producto de sus desviaciones típicas.

```
?to corr :cov :sd1 :sd2
>op / :cov (* :sd1 :sd2)
>end
corr defined
?corr 4 .5 8
1
?
```

La correlación no produce más que un único valor como resultado, pero la regresión produce dos: la constante de regresión y el coeficiente de regresión.

El coeficiente de regresión se determina calculando la covarianza de las dos listas de valores y dividiéndola por la varianza de la lista que contiene los valores del eje x. Realmente, el coeficiente de regresión no es más que la pendiente de la recta cuya ecuación es la siguiente:

```
prevy = regconst + regcoef * x
```

en la que **prevy** representa el valor Y previsto para cada X, mientras que **regconst** y **regcoef** corresponden a los coeficientes de igual nombre que definimos a continuación. La ecuación anterior define una recta que se conoce como "recta de regresión".

El procedimiento que permite calcular el coeficiente de regresión es el siguiente:

```
?to regcoef :cov :var
>op :cov / :var
>end
regcoef defined
?
```

La constante de regresión se obtiene trabajando con la ecuación de regresión, de forma que sea igual a la media de los valores del eje X multiplicada por el coeficiente de regresión y restando al resultado obtenido la media de los valores del eje Y. La constante de regresión representa el punto en que la recta de regresión corta al eje Y; el procedimiento para calcularla es el siguiente:

```
?to regconst :my :mx :rcoe
>op - :my (* :mx :rcoe)
>end
regconst defined
?
```

Puede combinarse estos dos procedimientos para obtener otro de regresión completo:

```
?to regres :my :mx :varx :cov
>local "coef
>make "coef regcoef :cov :varx
>op se (regconst :my :mx :coef) :coef
>end
regres defined
?regres 2 3 3 6
[-4 2]
?
```

Fijese en la utilización de la instrucción **se** para imprimir en pantalla una lista de dos valores. En este caso los paréntesis sólo sirven para delimitar el procedimiento **regconst** y sus parámetros, de forma que la lectura de la definición del procedimiento resulte más sencilla. LOGO realmente no los necesita. A continuación vamos a definir un procedimiento que calcule el coeficiente de correlación, la constante de regresión y el coeficiente de regresión en este orden.

```
?to correg :cov :vary :varx :my :mx
>op se (corr :cov raiz2 :vary raiz2 :varx)
(regres :my :mx :varx :cov)
>end
correg defined
?correg 6 48 3 2 2
[0.5 -2 2]
?
```

Observe detenidamente cómo hemos dividido todo el sistema estadístico en módulos. Hasta ahora no hemos definido ningún procedimiento que sea en sí largo o complicado; sin embargo el resultado final es muy potente.

10.7 Sistemas estadísticos con dos variables

El sistema estadístico con dos variables toma como datos dos listas de valores, dando como resultado otra lista cuyos elementos son los siguientes:

Número	Elemento
1	Coeficiente de correlación
2	Constante de regresión
3	Coeficiente de regresión
4	Covarianza
5	Desviación típica de los valores del eje Y
6	Desviación típica de los valores del eje X
7	Media de los valores del eje Y
8	Media de los valores del eje X

El procedimiento que calcula todos estos valores es el siguiente:

```
?to estadist :listy :listx
>(local "my "mx "vary "varx "cov "cr)
>make "my media :listy
>make "mx media :listx
>make "vary varianza :listy :my
>make "varx varianza :listx :mx
>make "cov covar :listy :listx :my :mx
>make "cr correg :cov :vary :varx :my :mx
>op (se :cr :cov (raiz2 :vary) (raiz2 :varx)
 :my :mx)
>end
estadist defined
?estadist [1 2 3 4 5] [10 8 6 4 2]
[-1 6 -0.5 -4 1.4142135623731 2.82842712474619
 3 6]
?
```

No es necesario seguir profundizando en el tema de la regresión y la correlación, puesto que ya hemos conseguido el objetivo fundamental que nos propusimos al principio del capítulo: demostrar la importancia de los sistemas modulares en las aplicaciones numéricas.

Dejo a los más interesados en este campo la tarea de dibujar los gráficos estadísticos y las líneas de regresión.

Capítulo 11

Tratamiento de matrices

El LOGO que emplean los ordenadores CPC 464/664 no es un lenguaje muy rápido, ya que, para conseguir un lenguaje tan sumamente flexible, se necesita un intérprete bastante largo que hace que la ejecución de los procedimientos sea más lenta.

Una de las tareas de programación más ardua y pesada es la realización de rutinas y procedimientos que permitan trabajar con matrices. En este capítulo vamos a dedicarnos a desarrollar varios de estos procedimientos; LOGO facilita bastante el trabajo. En realidad muchas de las ideas en que nos vamos a apoyar proceden de otro lenguaje de programación muy importante: el APL. Recomiendo a los interesados en las técnicas de programación que investiguen a fondo este potente lenguaje.

11.1 La matriz como tabla

En primer lugar vamos a dejar bien claro qué entendemos por matriz. Una matriz no es más que una tabla de datos.

En general podemos decir que cualquier disposición de datos en forma rectangular es una matriz, por ejemplo cualquier conjunto de datos clasificados por medio de índices. Las matrices no son más que estructuras de datos, así que el primer problema a resolver es cómo vamos a guardar en la memoria los contenidos de una matriz. Consideremos por ejemplo la matriz **A**:

1 2 3
4 5 6

Como puede verse, las matrices no tienen por qué ser complicadas. En este caso hay dos filas y tres columnas, seis elementos en total.

Las matrices tampoco tienen por qué tener una disposición determinada. Puede cambiarse las filas por columnas, con lo que obtendríamos una variante de la matriz anterior a la que llamaremos **B**:

1 4
2 5
3 6

Realmente estas matrices no son totalmente distintas; ambas guardan una cierta relación: son matrices traspuestas.

Algunos lenguajes de programación (como es este caso) distinguen entre filas y columnas por la forma de guardar las matrices en la memoria. En general siempre hay que tener presente que el lenguaje no trabaja con la matriz en sí, sino con sus elementos. No podemos referirnos a la matriz como un bloque, sólo podemos hablar de elementos especificando la fila y columna de cada uno de ellos. Por ejemplo, el número **6** es el elemento **A(3,2)** de la primera matriz, o el elemento **B(2,3)** de la segunda.

En todos los casos, el orden de los índices encerrados entre paréntesis es **número de fila, número de columna**.

Aunque hay lenguajes que permiten trabajar con matrices en conjunto, lo más habitual es que el tamaño de las mismas esté limitado y que sea necesario emplear procedimientos especiales para operar con ellas. Es muy interesante, por tanto, disponer de un conjunto de procedimientos en LOGO para aplicar en todos los casos y que sean suficientemente claros en su base conceptual.

Ya se ha dicho que una matriz es una tabla de datos, es decir, una lista de números múltiple en la que cada fila es a su vez una sublista de números. De esta manera es factible guardar en la memoria la matriz **A** de la siguiente forma:

```
?make "a [[1 2 3] [4 5 6]]
?
```

En este ejemplo cada fila es una sublista; las dos filas constituyen una lista completa que es la matriz. Éste es el método que se emplea en LOGO con más frecuencia, pero vamos a plantearnos un pequeño problema: ¿cómo pasar del ejemplo anterior a la matriz **B**?

```
?make "b [[1 4] [2 5] [3 6]]
```

Este método de almacenamiento de matrices no me gusta nada; creo que no es elegante ni consistente. Hay que reconocer que con él la tarea de leer las filas se hace muy sencilla, ya que no son más que sublistas, pero perdemos totalmente la integridad de las columnas.

Pienso que he encontrado una solución mejor.

11.2 La matriz como propiedad

Cuando empezamos a dar vueltas en la cabeza a la idea de las matrices en LOGO, llegamos invariablemente a la conclusión de que el hecho de que unos datos constituyan una matriz representa una propiedad distinta de las que hemos tratado hasta ahora (llegué a esta idea con la colaboración de Chris Lusby-Taylor); es precisamente esta propiedad la que debemos aplicar a los datos en sí.

El paso siguiente fue decidir cómo dar forma a esta propiedad. Es en este punto donde he empleado algunas nociones de APL de David Weatherby, de I P Sharp Associates. De él surgió la idea de que una matriz no es más que una lista de elementos a la que aplicamos un operador denominado "tamaño", es decir, para pasar de la matriz **A**:

1 2 3
4 5 6

a una matriz **C** de la forma

1 2
3 4
5 6

lo único que hay que hacer es modificar el tamaño de la matriz.

Las dos matrices proceden de la misma lista de números; la única diferencia aparente entre ambas está en el cambio de tamaño. Es evidente que, para obtener un resultado parecido, en LOGO son necesarios unos cuantos procedimientos que permitan escribir:

```
?make "a [1 2 3 4 5 6]
?tam "a [2 3]
?make "c :a
?tam "c [3 2]
?matrlist "a
1 2 3
4 5 6
?matrlist "c
1 2
3 4
5 6
?
```

Lo único que hemos hecho ha sido introducir una lista de valores en **a**, indicando que el tamaño de **a** es **[2,3]**; a continuación hemos copiado la misma lista de valores en **c**, pero esta vez el tamaño de la matriz es **[3,2]**. Por último, el procedimiento **matrlist** permite imprimir en pantalla cualquier lista con forma de matriz, teniendo en cuenta el tamaño de la misma.

Vamos a definir cada uno de los procedimientos empleados. Empezaremos con **tam**:

```
?to tam :nom :dim
>pprop :nom ".dim :dim
>end
tam defined
?
```

El primer parámetro de **tam** es el nombre del objeto que tiene la propiedad que estamos definiendo, mientras que el segundo es el valor de la propiedad en sí. La primitiva **pprop** permite dar el valor **:dim** a la propiedad **".dim**. En esta propiedad se basa todo el sistema, y la empleará el procedimiento **matrlist**.

Vamos a examinar detenidamente **matrlist** para profundizar en el sistema:

```
?to matrlist :nom
>(local "f "ctr)
>make "f first gprop :nom ".dim
>bucle "ctr 1 :f [pr fila :nom :ctr]
>end
matrlist defined
?
```

Este procedimiento hace intervenir a su vez a los procedimientos **bucle** y **fila**, pero antes de definir estos últimos vamos a ver cómo funciona **matrlist**.

Este procedimiento emplea dos variables locales: **f** y **ctr**. La primera de ellas indica cuántas filas tiene la matriz representada con **:nom**; fíjese que hemos empleado la primitiva **grop** para atribuir el valor de **".dim** a **:nom**. Recuerde así mismo que **".dim** contiene la lista con las dimensiones de la matriz. La segunda variable, **ctr**, sirve de contador del procedimiento **bucle**; su valor va desde **1** hasta **f**.

El procedimiento **bucle** imprime el número de fila, **:ctr**, del objeto **:nom** teniendo en cuenta su tamaño, es decir, **".dim**.

Tras esto ya pueden explicarse los procedimientos **fila** y **bucle**.

11.3 Bucles controlados

La primitiva **repeat** de Dr LOGO permite realizar bucles, pero no controlarlos, dado que su duración no está ligada al contenido de la lista de instrucciones. A continuación está definido el procedimiento **bucle**, que proporciona un mayor control.

El procedimiento es el siguiente:

```
?to bucle :var :ini :fin :lista_bucle
>if :ini > :fin [stop]
>make :var :ini
>run :lista_bucle
>bucle :var :ini + 1 :fin :lista_bucle
>end
bucle defined
?
```

He definido el procedimiento de forma recurrente porque resulta algo más cómodo y atractivo, y al mismo tiempo hace mínimo el número de variables locales y de ins-

trucciones **make** necesarias. Sus parámetros son **:var**, que representa el nombre de la variable del bucle y que se emplea en las instrucciones de **bucle_lista**, **:ini**, que representa el valor inicial, y **:fin**, que representa el valor final. El bucle se detiene en cualquier momento en que el valor inicial sobrepase al valor final.

Este procedimiento puede ser de utilidad en muchos otros casos y se asemeja ligeramente a los bucles FOR de otros lenguajes. El procedimiento **matrlist** hace uso del procedimiento **bucle** para pasar por todas las filas de una en una, siendo **ctr** la variable que representa en cada momento el número de fila. La numeración de las filas va de **1** a **f** en sentido creciente. Habitualmente trabajamos con bucles en los que el contador va creciendo hasta alcanzar su valor máximo; no obstante también puede ser de interés el procedimiento **#bucle**, en el que el contador del bucle va tomando valores descendentes.

```
?to #bucle :var :ini :fin :lista_bucle
>if :ini > :fin [stop]
>make :var :ini
>run :lista_bucle
>#bucle :var :ini - 1 :fin :lista_bucle
>end
#bucle defined
?
```

Este procedimiento hace que el contador **:var** descienda desde el valor **:ini** hasta el valor **:fin** de unidad en unidad. Vamos a emplearlo en los procedimientos **fila** y **col**.

```
?to fila :nom :f
>(local "temp "ctr)
>make "ctr item 2 gprop :nom ".dim
>make "temp [ ]
>#bucle "ctr :ctr 1 [make "temp se (elem :nom
:f :ctr) :temp]
>op :temp
>end
fila defined
?to col :nom :c
>(local "ctr "temp)
>make "ctr first gprop :nom ".dim
>make "temp [ ]
>#bucle "ctr :ctr 1 [make "temp se (elem :nom
:ctr :c) :temp]
>op :temp
>end
col defined
?
```

El procedimiento **fila** emplea dos parámetros: **:nom**, que contiene el nombre de la matriz, y **:f**, que contiene el número de la fila.

A su vez, la variable local **temp** sirve para almacenar el contenido de la fila que nos ocupa temporalmente en cada pasada del bucle. Por su parte, **ctr** toma al principio del procedimiento como valor el número de columnas de la matriz (el segundo elemento de la lista **".dim**). Más adelante, al llegar al procedimiento **#bucle**, la variable **ctr** pasa a ser el contador del bucle. El procedimiento **#bucle** crea la variable local **temp** con elementos de la lista tomados desde la derecha de la misma y añadidos a la izquierda de **temp** por medio de **se**.

El procedimiento construye la primera fila de la matriz **a** de la siguiente manera: **[]**, **[3]**, **[2 3]** y por último **[1 2 3]**. Si hubiésemos empleado **bucle** habríamos obtenido la misma fila en orden contrario al anterior, añadiendo elementos con **se** a la derecha de la lista **temp**. El procedimiento **col** es similar al anterior; vuelve a hacer uso de **#bucle**, lo que quiere decir que cada columna se forma de abajo hacia arriba. Estos procedimientos permiten hacer referencia a una fila o a una columna determinada de la matriz con la que estemos trabajando, con lo que el tratamiento de matrices queda muy simplificado.

11.4 Cómo elegir un elemento

Los procedimientos que acabamos de ver hacen uso de uno básico, **elem**, en el que se basan prácticamente todos los que facilitan de alguna forma el tratamiento de matrices.

En pocas palabras, **elem** emplea dos parámetros, un número de fila y un número de columna, y da el elemento correspondiente de una matriz determinada. La selección del procedimiento se realiza como si estuviésemos trabajando en APL:

```
?make "z [1 2 3 4]
?tam "z [3 5]
?matrlist "z
1 2 3 4 1
2 3 4 1 2
3 4 1 2 3
?
```

En este caso, la lista de valores, es decir, la propiedad **.APV** de la variable **z**, no tiene más que cuatro elementos, pero al haber definido el tamaño de **z** como **[3 5]**, la matriz debe tener quince elementos.

Desde luego los dos números de elementos no concuerdan, quince es considerablemente superior a cuatro, así que podemos hacer dos cosas: o bien enviar un mensaje de error, o bien aumentar la lista de cuatro elementos hasta quince. Esto último es lo que realiza APL. En el ejemplo anterior podemos ver claramente cómo hemos aumentado la lista de elementos: al llegar al quinto elemento de la matriz, es decir,

a la quinta columna de la primera fila, volvemos al primer elemento de la lista de valores, repitiendo el proceso tantas veces como sea preciso.

Por el contrario, si emplea la siguiente alternativa:

```
?tam "z [2 1]
?matrlist "z
1
2
?
```

puede observar que si la matriz solicita menos elementos que los contenidos en la lista, LOGO no toma más que los que le hacen falta. Siempre queda el recurso de modificar la definición de **elem**, de forma que el número de elementos de la lista coincida forzosamente con el número de elementos de la matriz.

A continuación se dan las definiciones de **elem** y de otro procedimiento denominado **mod**:

```
?to elem :nom :f :c
>(local "ctr "tc "apv)
>make "apv gprop :nom ".APV
>make "tc item 2 gprop :nom ".dim
>make "ctr 1 + mod ((:f - 1) * :tc +
:c - 1) (count :apv)
>op item :ctr :apv
>end
elem defined
?to mod :num :div
>op :num - :div * int (:num / :div)
>end
mod defined
?int 15/4
3
?4 * 15/4
12
?15 - 4 * int 15/4
3
?mod 15 4
3
?
```

Vamos a comenzar con **mod**. Este procedimiento da el resto de la división entera del primer dato por el segundo. Para ver cómo funciona no hay más que fijarse en las líneas que siguen a la definición del procedimiento. **int** es una primitiva que realiza la división entera de dos números.

El procedimiento **elem** hace uso de tres variables locales: **ctr**, en la que introduci-

mos el número del elemento de la lista que vamos a elegir, **tc**, en la que introducimos el número de columnas de la matriz y que ya hemos visto que es el segundo elemento de la lista **".dim**, y por último **apv**, en la que guardamos el valor de la propiedad **.APV** de la matriz **:nom**. El funcionamiento del procedimiento **elem** merece un estudio detenido, en particular la línea **make "ctr 1 + . . .**

El procedimiento **elem** es muy fácil de utilizar; no hay más que introducir el nombre de la matriz y los números de fila y columna del elemento que interesa. De esta forma, mientras en otros lenguajes pedimos el elemento **a[f,c]**, en LOGO se solicita el elemento **elem "a :f :c**. Al examinar los siguientes procedimientos podrá apreciar qué significa esta simplificación, ya que aporta un método para "indexar" matrices.

11.5 Trasposición de matrices

El primer procedimiento que vamos a ver, que también es el más sencillo de concepto, determina la matriz traspuesta de una dada, teniendo en cuenta que ahora ya contamos con un método que permite establecer la estructura de datos de una matriz con **.APV** y **.dim**.

Estudie la siguiente secuencia:

```
?crear "a [1 2 3 4] [3 5]
?matrlist "a
1 2 3 4 1
2 3 4 1 2
3 4 1 2 3
?copiar "c "a
?matrlist "c
1 2 3 4 1
2 3 4 1 2
3 4 1 2 3
?trans "b "a
?matrlist "b
1 2 3
2 3 4
3 4 1
4 1 2
1 2 3
?plist "b
[.dim [5 3] .APV [1 2 3 2 3 4 3 4 1 4 1 2 1]
2 3]
?
```

El primer dato del procedimiento **crear** es el nombre de la matriz, el segundo es la lista de valores y el tercero es el tamaño o dimensiones de la propia matriz. El procedimiento **matrlist** permite ver la matriz resultante.

El procedimiento **copiar**, como su propio nombre indica, permite copiar los contenidos de una matriz en otra diferente. En el ejemplo anterior se ha empleado para crear la matriz **c** a partir de la **a**. Por su parte, el procedimiento **trans** permite determinar la matriz traspuesta de una dada, como se puede comprobar por medio de **matrlist** o pidiendo una lista de las propiedades de la matriz con la primitiva **plist**. Observe que al construir una matriz con **trans** la lista de valores se adapta exactamente a la nueva estructura de datos, es decir, la lista tiene tantos datos como la matriz.

Hay dos procedimientos que se emplean para crear matrices; son los siguientes:

```
?to crear :nom :apv :dim
>make :nom :apv
>tam :nom :dim
>end
crear defined
?to copiar :dest :orig
>make :dest gprop :orig ".APV
>tam :dest gprop :orig ".dim
>end
copiar defined
?
```

No debe tener ningún problema para comprenderlos; recuerde que al principio del capítulo se definió el procedimiento **tam**. Es posible que el tercer procedimiento, **trans**, sea un poco más difícil de comprender.

```
?to trans :dest :orig
>make :dest transval :orig
>tam :dest :invertir :orig
>end
trans defined
?to transval :nom
>(local "fc "temp)
>make "fc gprop :nom ".dim
>make "temp [ ]
>bucle "fc 1 (item 2 :fc) [make "temp se :temp
col :nom :fc]
>op :temp
>end
transval defined
?to invertir :nom
>local "mid.
>make "mid. gprop :nom ".dim
>op se item 2 :mid. item 1 :mid.
>end
invertir defined
?
```

Aunque estos procedimientos son un poco más complejos, no creo que necesite mi ayuda para comprenderlos. Conviene tener presente en todo momento que las columnas de la matriz **origen** pasan a ser las filas de la matriz **destino**.

11.6 Combinación de matrices

Los dos métodos de combinación de matrices que daré a continuación se basan en lo que considero que son las desventajas del APL, aunque probablemente los programadores de APL no estén de acuerdo conmigo. No es mi intención dar muchas explicaciones sobre estos procedimientos; prefiero aplicarlos a unos cuantos ejemplos y dejar que el lector descubra cómo hemos obtenido los resultados en cada caso.

La siguiente secuencia ilustra el fucionamiento del procedimiento **matrop**.

```
?crear "a [1 2 3 4] [2 3]
?matrlist "a
1 2 3
4 1 2
?matrop "b "+ "a "a
?matrlist "b
2 4 6
8 2 4
?matrop "b "* "a "a
?matrlist "b
1 4 9
16 1 4
?matrop "c "_ "b "a
?matrlist "c
0 2 6
12 0 2
?trans "b "a
?matrlist "b
1 4
2 1
3 2
?matrop "c "* "b "a
dimensiones incompatibles
?
```

El mensaje de la última línea indica que se puede emplear **matrop** con matrices de tamaños diferentes. En esencia, **matrop** toma dos matrices (**m1** y **m2**) y realiza con ellas, elemento a elemento, la operación que determina el **oper**ador que asigne al segundo parámetro, almacenando el resultado final en la matriz que indique en el primer parámetro. El segundo parámetro de **matrop** puede ser cualquier procedi-

miento que haya definido usted mismo, siempre y cuando actúe sobre dos parámetros, que serán los dos elementos de la matriz.

A continuación puede ver la definición de **matrop**:

```
?to matrop :res :oper :m1 :m2
>if not = (gprop :m1 ".dim) (gprop :m2 ".dim)
[pr [dimensiones incompatibles] stop]
>(local "fc "filas "cols "fi "ci "temp)
>make "temp [ ]
>make "fc gprop :m1 ".dim
>make "filas item 1 :fc
>make "cols item 2 :fc
>bucle "fi 1 :filas [bucle "ci 1 :cols [make
"temp (se :temp run (se :oper elem :m1 :fi
:ci :fc elem :m2 :fi :ci :fc))]]
>pprop :res ".dim :fc
>make :res :temp
>end
matrop defined
?
```

Este procedimiento se parece bastante al procedimiento **combinar**, que ya habíamos utilizado en el sistema estadístico. Observe cómo se ha empleado la primitiva **run** para ejecutar una lista de instrucciones.

El último procedimiento que vamos a ver es bastante más técnico que los anteriores, pero no está explicado a fondo; aquellos lectores que deseen profundizar un poco en el tema de las matrices pueden consultar el libro ''Teach Yourself: New Mathematics'', de L C Pascoe. Este procedimiento calcula el producto de matrices cuando el segundo parámetro es "*.

Antes de pasar a definir el procedimiento, una pequeña introducción:

```
?crear "a [1 2 3] [1 3]
?trans "b "a
?matrlist "b
1
2
3
?matrlist "a
1 2 3
?matropx "c "* "b "a
?matrlist "c
1 2 3
2 4 6
3 6 9
```

```
?matropx "d "* "a "b
?matrlist "d
14
?tam "a [2 3]
?matrlist "a
1 2 3
1 2 3
?trans "b "a
?matrlist "b
1 1
2 2
3 3
?matropx "c "* "a "b
?matrlist "c
14 14
14 14
?matropx "d "* "b "a
?matrlist "d
2 4 6
4 8 12
6 12 18
?crear "a [1 2 3 3 2 1] [2 3]
?matrlist "a
1 2 3
3 2 1
?trans "b "a
?matrlist "b
1 3
2 2
3 1
?matropx "c "* "a "b
?matrlist "c
14 10
10 14
?
```

Las definiciones de los procedimientos son las siguientes:

```
?to matropx :res :oper :m1 :m2
>if not = (item 1 gprop :m1 ".dim) (item 2
gprop :m2 ".dim) [pr [dimensiones
incompatibles] stop]
>(local "fc "filas "cols "fi "ci "temp)
>make "temp [ ]
>make "filas item 1 gprop :m1 ".dim
>make "cols item 2 gprop :m2 ".dim
```

```
>bucle "fi 1 :filas [bucle "ci 1 :cols [make
"temp (se :temp sigma combinar :oper
fila :m1 :fi col :m2 :ci)]]
>pprop :res ".dim se :filas :cols
>make :res :temp
>end
matropx defined
?to sigma :l
>(local "temp "ctr)
>make "temp 0
>make "ctr count :l
>bucle "ctr 1 :ctr [make "temp :temp + item
:ctr :l]
>op :temp
>end
sigma defined
?to combinar :oper :l1 :l2
>if not = (count :l1) (count :l2) [pr [Error
en las dimensiones] stop]
>(local "temp "i)
>make "temp [ ]
>bucle "i 1 count :l1 [make "temp se :temp run
(se :oper item :i :l1 item :i :l2)]
>op :temp
>end
combinar defined
?
```

Observe que los procedimientos **combinar** y **sigma** son diferentes de los definidos en el sistema estadístico. Es interesante compararlos.

Capítulo 12

PROLOGO

En este capítulo voy a sentar los principios de una base de datos que tiene ciertas similitudes con PROLOG, un lenguaje de la quinta generación. Desde luego no es mi intención cubrir todas las funciones de PROLOG, pero espero demostrar que una versión de LOGO relativamente pequeña, consigue efectos muy interesantes.

Lo que intento es conseguir una base de datos que sea funcionalmente equivalente a PROLOG, pero sería inútil intentar imitar en todos sus detalles a este lenguaje. Por ejemplo, la versión de PROLOG para micros emplea el siguiente sistema para añadir datos nuevos a otros ya existentes:

```
&.add(enrique padre_de isabel_II)
```

mientras que el método funcionalmente equivalente en LOGO es:

```
?pprop "enrique "padre_de "isabel_II
```

De forma similar trabajaré en el resto de los casos.

12.1 Hechos y relaciones

En su nivel más sencillo, PROLOG trabaja con "hechos", afirmaciones del tipo "Sujeto Relación Objeto", o en términos de LOGO "Objeto Propiedad Contenido". Al registrar el hecho "Enrique es padre_de Isabel_II", realmente se añade dos datos a la base; para ello se emplea el procedimiento **add**:

```
?to add :suj :rel :obj
>pprop :suj :rel :obj
>pprop :obj word "era_ :rel :suj
>end
add defined
?add "enrique "padre_de "isabel_II
?plist "enrique
[padre_de isabel_II]
?plist "isabel_II
[era_padre_de enrique]
?
```

Este procedimiento deja sin resolver un pequeño detalle:

```
?add "enrique "padre_de "eduardo
?plist "enrique
[padre_de eduardo]
?
```

La respuesta a la instrucción **plist** no es completa, ya que debería decir que **enrique** tuvo dos hijos; así que en lugar de emplear la primitiva **pprop** debemos emplear el procedimiento **adprop**, que añade información a las propiedades ya existentes:

```
?to adprop :nom :prop :conten
>pprop :nom :prop se :conten gprop :nom
:prop
>end
adprop defined
?to add :suj :rel :obj
>adprop :suj :rel :obj
>adprop :obj word "era_ :rel :suj
>end
add defined
?add "enrique "padre_de "eduardo
?add "enrique "padre_de "isabel_II
?plist "enrique
[padre_de [isabel_II eduardo]]
?add "enrique "era "varon
?plist "enrique
[era [varon] padre_de [isabel_II eduardo]]
?
```

Pero aún queda por resolver un detalle; si escribimos:

```
?add "enrique "padre_de "eduardo
?gprop "enrique "padre_de
[eduardo isabel_II eduardo]
?
```

tenemos que encontrar algún sistema para evitar que se duplique la información que hay registrada al introducir dos veces un dato. La solución es introducir una pequeña modificación en el procedimiento **adprop**:

```
?to adprop :nom :prop :conten
>if yaprop :conten gprop :nom :prop [stop]
>pprop :nom "CLAVE [*]
>pprop :nom :prop se :conten gprop :nom
:prop
```

```
>end
adprop defined
?to yaprop :elem :enlista
>if 0 = count :enlista [op "FALSE]
>if :elem = first :enlista [op "TRUE] [op
yaprop :elem bf :enlista]
>end
yaprop defined
?
```

El procedimiento **yaprop** existe en forma de primitiva en algunas versiones de LOGO. Su resultado es **TRUE** en caso de que el **:elem**ento dado pertenezca a la lista **:enlista**. La línea que incluye la palabra **"CLAVE** no es imprescindible; su única utilidad real aparece a la hora de diferenciar un objeto dentro de la base de datos. Más adelante se la empleará para guardar la base de datos como si se tratase de un valor.

12.2 Cómo crear una base de datos

Una vez definidos los procedimientos anteriores, conviene grabarlos en disco y salir de LOGO. Con ello, al volver a cargar LOGO tendremos libre el área de trabajo y podremos volver a empezar desde cero. Vamos a emplear como ejemplo de base de datos la FAMILIA del rey Enrique VIII de Inglaterra, definida por K L Clark, F G McCabe y J R Ennals en sus «Fundamentos de micro-PROLOG». El procedimiento es el siguiente:

```
?add "enrique_VII "padre_de "enrique
?add "enrique "padre_de "isabel_II
?add "enrique "padre_de "eduardo
?add "catalina "madre_de "isabel_II
?add "ana "madre_de "isabel_II
?add "juana "madre_de "eduardo
?add "enrique_VII "era "varon
?add "enrique "era "varon
?add "eduardo "era "varon
?add "isabel_I "era "mujer
?add "catalina "era "mujer
?add "maria "era "mujer
?add "isabel_II "era mujer
?add "ana "era "mujer
?add "juana "era "mujer
?
```

En el libro arriba citado los autores solicitan a PROLOG una lista de todos los casos en los que exista la relación **madre_de**. El equivalente en LOGO es solicitar to-

dos los elementos que tengan la propiedad **madre_de**, junto con el propio objeto, la propiedad y su contenido. Para ello vamos a definir el siguiente procedimiento:

```
?to listacon :prop
>conprop :prop glist :prop
>end
listacon defined
?to conprop :prop :enlista
>if emptyp :enlista [stop]
>(pr first :enlista :prop (list gprop first
:enlista :prop))
>conprop :prop bf :enlista
>end
conprop defined

?listacon "madre_de
catalina madre_de [maria]
ana madre_de [isabel_II]
juana madre_de [eduardo]
isabel_I madre_de [enrique]
?listacon "padre_de
enrique padre_de [eduardo isabel_II]
enrique_VII padre_de [enrique]
```

Ahora suponga que queremos saber si **enrique** fue **padre_de isabel_II** o si **catalina** fue **madre_de eduardo**; nada más fácil. Fíjese que en el segundo ejemplo hemos invertido los términos de la pregunta:

```
?yaprop "isabel_II gprop "enrique "padre_de
TRUE
?yaprop "catalina gprop "eduardo "era_madre_de
FALSE
```

Y ahora, si queremos saber quién era la **madre_de maria** escribimos:

```
?gprop "maria "era_madre_de
[catalina]
?to condos :l1 :l2
>if emptyp :l1 [op []]
>if yaprop first :l1 :l2 [op se first :l1
condos bf :l1 :l2] [op bf :l1 :l2]
>end
condos defined
?condos gprop "enrique_VII "padre_de gprop "isabel_II
"era_padre_de
[enrique]
?
```

Las últimas líneas indican que **enrique** tuvo por padre a **enrique VII** y que, a su vez, era el padre de **isabel_II**.

Todos estos ejemplos proceden del libro de PROLOG citado anteriormente, y permiten apreciar que LOGO puede responder perfectamente a una serie de consultas sencillas propias de PROLOG. Antes de seguir adelante habrá que grabar en disco la base de datos.

12.3 Cómo grabar en disco una base de datos

Al grabar en disco el área de trabajo, LOGO graba automáticamente todas las propiedades, ya sea en relación con **make** o con **pprop**. Esto quiere decir que para grabar en disco la base de datos no hay más que grabar el área de trabajo en la forma habitual. Para ver cómo ha quedado registrada la base de datos puede usted salir a CP/M y sacar en pantalla una copia del fichero correspondiente.

Suponga que queremos almacenar la base de datos, o parte de ella, en una variable; por ejemplo:

```
?glist "CLAVE
[catalina enrique isabel_I ana juana maria
varon eduardo mujer isabel_II enrique_VII]
?
```

La primitiva **glist** proporciona todos los objetos que tengan la propiedad definida en su parámetro. Esta es la causa por la que hemos incluido en **adprop** la línea con **CLAVE**; ahora podemos distinguir unos objetos de la base de datos de otros según el contenido de la propiedad **CLAVE**. También podemos asignar la base de datos a un objeto con las siguientes instrucciones:

```
?make "parte soloparte
?
```

que se encargan de almacenar la base en el objeto **parte**. Esto permite examinar la base de datos con **:parte** o con **grop "parte ".APV**. A continuación está definido el procedimiento **soloparte**, que se basa en la primitiva **glist**:

```
?to soloparte
>local "temp
>make "temp []
>soloaux glist "CLAVE
>op :temp
>end
soloparte defined
?to soloaux :en
>if emptyp :en [stop]
```

```
>make "temp fput first :en fput plist first
:en :temp
>soloaux bf :en
>end
soloaux defined
?
```

El primer procedimiento, **soloparte**, no es más que una introducción a **soloaux**, en el que se define la variable local **temp**. El segundo procedimiento, **soloaux**, recibe como parámetro una lista de objetos y coloca al principio de la lista **temp** las propiedades del primer elemento de la lista dato. A continuación el procedimiento se llama a sí mismo en forma recurrente, empleando como parámetro el resto de los elementos de la lista inicial.

Al grabar el área de trabajo en disco, se graba automáticamente toda la información que contiene **parte**. Es probable que el hecho de almacenar la base de datos en una variable no sea de gran utilidad práctica, pero ilustra bastante bien todo lo que se puede conseguir procesando listas.

12.4 Cómo recuperar la base de datos

Una vez que hemos borrado el área de trabajo saliendo a CP/M y volviendo a LOGO, podemos volver a cargar el área de trabajo que habíamos grabado en disco, con lo cual tendremos en memoria al mismo tiempo un objeto denominado **parte** que contiene la base de datos y la propia base de datos en sí. Necesitamos "desempaquetar" cada elemento individual contenido en **parte**.

Para proceder al desempaquetado, usaremos dos procedimientos que hasta cierto punto son las imágenes reflejas de los utilizados para almacenar la base de datos:

```
?to separar :en
>if emptyp :en [stop]
>sepaux first :en item 2 :en
>separar bf bf :en
>end
separar defined
?to sepaux :nom :en
>if emptyp :en [stop]
>pprop :nom first :en item 2 :en
>sepaux :nom bf bf :en
>end
sepaux defined
?separar :parte
?plist "parte
[.APV [enrique_VII [era [varon] padre_de
[enrique] CLAVE [*]] isabel_II ...
```

Con todos estos procedimientos ya estamos en condiciones de grabar en disco y recuperar nuestra base de datos. Como ya he indicado, es posible que estas rutinas no tengan ninguna utilidad práctica en este caso, pero de todas formas ilustran bastante bien un grupo de técnicas de programación que son muy útiles.

En resumen, ya sabemos cómo establecer reglas y comprobar si se cumplen. Por ejemplo, el hermanastro de una persona es otra persona que tenga su mismo padre o su misma madre, pero no ambos porque en este caso sería su hermano. Queda por último otra posibilidad, que es definir estas reglas en forma de propiedades.

El desarrollo de todas estas posiblidades quedan en manos del lector. Para comenzar puede buscar la manera de borrar **parte** de la base de datos. Es muy sencillo borrar propiedades con la primitiva **remprop**, pero no es tan sencillo borrar **parte**. De todas formas, puede conseguirse...

Apéndice A

LOGO: Órdenes y primitivas

Se incluye en este apéndice un juego completo de las órdenes y primitivas de la versión de LOGO que emplean los ordenadores Amstrad, junto con ejemplos que ilustran el funcionamiento de cada una. Muchas de ellas han aparecido con anterioridad a lo largo de la Guía, pero no han sido tratadas todas con el debido detalle.

Es interesante que el lector practique a lo largo de este apéndice el uso de las primitivas que no han aparecido hasta ahora.

Todas las instrucciones aparecen en orden alfabético, es decir, según los códigos ASCII de los caracteres que componen cada una de ellas, no agrupadas según las funciones que realizan. He seguido este criterio porque habitualmente lo que recordamos con más claridad es el nombre de las instrucciones. Si usted quiere investigar sobre una orden, generalmente es su función lo que le será de interés.

*

Este operador da el resultado de multiplicar los dos datos que le acompañan. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 * 3
6
?* 2 2
4
?
```

+

Este operador da el resultado de sumar los dos datos que le acompañan. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar la operaciones.

```
?2 + 3
5
?+ 2 2
4
?
```

–

Este operador da el resultado de restar el dato que aparece a la izquierda del dato de la derecha. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 – 3
–1
?– 2 2
0
?
```

.APV

Se trata de una propiedad estándar del sistema que representa el valor de una variable global. Al solicitar el valor de **.APV** para una variable local se obtendrá el resultado nulo.

```
?make "a 9
?gprop "a ".APV
9
?to prueba
>local "b
>make "b 9
>op gprop "b ".APV
>end
prueba defined
?prueba
[]
?
```

Es posible emplear **.APV** para crear la primitiva **thing**, que contienen otras versiones de LOGO, por medio del siguiente procedimiento:

```
?to thing :nombre
>op gprop :nombre ".APV
>end
thing defined
?
```

pero recuerde que esto no funciona con variables locales por las razones vistas en el primer ejemplo.

.contents

Esta orden no lleva parámetros. Indica los contenidos del área de trabajo.

```
?.contents
[emptyp fput po ...]
?
```

El área de trabajo contiene todas las primitivas de LOGO, junto con los procedimientos y variables que define el ususario y los errores cometidos al introducir los datos. LOGO mantiene el nombre de los procedimientos en el área de trabajo aunque hayan sido borrados. Estos nombres son listas sin propiedades de ningún tipo; por tanto LOGO da por hecho que tales nombres no representan nada y los mantiene fuera de uso.

.DEF

Es una propiedad estándar del sistema y representa la definición de un procedimiento.

```
?gprop "prueba ".DEF
[[] [local "b] [make "b 9] [op gprop
"b ".APV]]
?
```

Ya hemos visto que, al borrar un procedimiento, LOGO mantiene el nombre del mismo en el área de trabajo, pero borra la propiedad **.DEF** que asocia dicho nombre con la definición de un procedimiento.

La lista obtenida como resultado contiene la definición del procedimiento en cierto orden un tanto especial. La primera sublista contiene los parámetros del procedimiento y cada una de las siguientes sublistas representa una línea de la propia definición. Es interesante comparar la definición de **prueba** que hemos visto al hablar de **.APV** con el resultado de **grop "prueba ".DEF**.

.deposit

Esta primitiva emplea dos parámetros; su función es colocar el valor del segundo de ellos en la posición de memoria que indica el primero. Véase también **.examine**.

```
?.examine 0
195
?.deposit 0 195
?.deposit O 195+1024
.deposit doesn't like 1219 as input
?
```

El valor del segundo parámetro debe ser inferior a 256, ya que cada posición de memoria contiene un byte y con un byte podemos representar números entre 0 y 255.

.examine

Esta primitiva emplea solamente un parámetro: el número correspondiente a una posición de memoria. Proporciona el valor que contiene dicha dirección.

```
?.examine 0
195
?
```

Véase asimismo la primitiva **.deposit**.

.PRM

Se trata de una propiedad estándar del sistema que proporciona la posición de memoria en que comienza la rutina en código de máquina de cualquier primitiva de LOGO que sea solicitada.

```
?gprop "emptyp ".PRM
3489
?
```

Teóricamente, este dato permite introducir modificaciones en las rutinas en código de máquina sin más que modificar el valor de la propiedad. También se puede emplear **.PRM** para obtener todas las primitivas que reconoce LOGO. Tenga en cuenta que si empleamos **.contents** para obtener el mismo resultado, es muy difícil saber exactamente cuáles son las primitivas en sí.

```
?glist ".PRM
[emptyp fput po dir op ...]
?
```

/

Este operador da el resultado de dividir el dato de la izquierda por el de la derecha. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 / 3
0.666666666666667
?/ 2 2
1
?
```

<

Este operador indica si el dato de la izquierda es menor que el de la derecha. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 < 3
TRUE
?< 2 2
FALSE
?"a < "b
TRUE
?< "a "B
FALSE
?
```

Cuando empleamos este operador con caracteres, el ordenador realiza la comparación de los códigos ASCII correspondientes.

=

Este operador indica si el dato de la izquierda es igual al de la derecha. Puede emplearse como prefijo o como infijo; en ambos casos suele ser útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 = 3
FALSE
?= "a "a
TRUE
?
```

>

Este operador indica si el dato de la izquierda es mayor que el de la derecha. Puede emplearse como prefijo o como infijo; en ambos casos suele útil emplear paréntesis para aclarar el orden en que se deben realizar las operaciones.

```
?2 > 3
FALSE
?> 2 2
FALSE
?"a > "b
FALSE
?> "a "B
TRUE
?
```

Cuando empleamos este operador con caracteres, el ordenador realiza la comparación de los códigos ASCII correspondientes.

and

Se trata de un operador prefijo que indica, por medio de **TRUE** o **FALSE**, si todos los datos que aparecen a continuación del mismo son ciertos. En caso de emplear más de dos datos es preciso encerrarlos, junto con la primitiva, entre paréntesis.

```
?and "TRUE "TRUE and "TRUE "FALSE
TRUE
FALSE
?(and "TRUE "TRUE "FALSE)
FALSE
?
```

ascii

Este operador solamente emplea un dato, habitualmente un carácter. Su resultado es el valor del código ASCII de dicho carácter.

```
?ascii "a
97
?ascii "abc
97
?
```

En caso de que el dato sea una palabra, el operador sólo tiene en consideración el primer carácter. Si el dato es una lista, LOGO responde con un mensaje de error.

```
?ascii [a]
ascii doesn't like [a] as input
?
```

bf

Abreviatura de **"but first"**. Proporciona todos los elementos de un objeto menos el primero.

```
?bf "abc bf [a b c] bf 12345
bc
[b c]
2345
?
```

En caso de que el objeto sea una lista formada por sublistas, **bf** trabaja de la siguiente manera:

```
?bf [[a b] [c d e] [f g h]]
[[c d e] [f g h]]
?bf bf [[a b] [c d e] [f g h]]
[[f g h]]
?bf bf bf [[a b] [c d e] [f g h]]
[ ]
?
```

bk

Abreviatura de "back". Esta primitiva hace que la tortuga retroceda un número de unidades igual al dato que la acompaña. Si este dato es negativo, la tortuga avanza en lugar de retroceder.

```
?bk 20
?bk -20
?
```

bl

Abreviatura de **"but last"**. Proporciona todos los elementos de un objeto menos el último.

```
?bl "abc bl [1 2 3] bl 12345
ab
[1 2]
1234
?
```

En caso de que el objeto sea una lista formada por sublistas, **bf** trabaja de la siguiente manera:

```
?bl [[a b] [c d e] [f g h]]
[[a b] [c d e]]
?bl bl [[a b] [c d e] [f g h]]
[[a b]]
?bl bl bl [[a b] [c d e] [f g h]]
[ ]
?
```

buttonp

Abreviatura de "button pressed". Esta primitiva lleva un parámetro que puede tomar los valores 0 o 1, y nos indica, por medio de **TRUE** o **FALSE**, si está pulsado el botón del joystick correspondiente al parámetro.

```
?buttonp 0
FALSE
?
```

Como consecuencia de la forma en que el joystick emula al teclado, su uso puede hacer aparecer caracteres en pantalla.

bye

Esta orden no lleva ningún parámetro. Su función es abandonar LOGO y volver a CP/M.

```
?bye
A>
```

catch

Esta primitiva emplea dos parámetros; el primero es el nombre o etiqueta que identifica a la instrucción **catch**, mientras que el segundo es una lista de sentencias a ejecutar. Cuando el ordenador encuentra en la instrucción **catch** una orden **throw** con la misma etiqueta, salta a la línea de instrucciones siguiente a **catch**.

```
?to prueba
>catch "si [otroproc]
>pr [se ha pulsado el boton]
>end
prueba defined

?to otroproc
>if buttonp 0 [throw "si]
>otroproc
>end
otroproc defined
?prueba
se ha pulsado el boton
?
```

En este caso el procedimiento **prueba** contiene la línea **catch "si (otroproc)**, que se encarga de ejecutar el procedimiento **otroproc**. LOGO conserva en memoria la etiqueta de **catch**, es decir, **si**. El procedimiento **otropoc** se encarga a su vez de comprobar si se ha pulsado el botón del joystick y, en caso contrario, vuelve a llamarse a sí mismo en forma recurrente, permaneciendo dentro de este bucle hasta que sea pulsado el botón.

Cuando se pulsa el botón, LOGO ejecuta la instrucción **throw "si**, que devuelve el control a la etiqueta **si** y por tanto a la sentencia **catch**. Observe que el ordenador sólo ejecuta la primitiva **catch** si la instrucción **throw** proviene de un procedimiento al que ha llamado la lista de instrucciones del propio **catch**. Finalmente, el control pasa a la sentencia que sigue a **catch**, en este caso **pr**.

char

Esta primitiva admite un número como parámetro y proporciona el carácter correspondiente al código ASCII. Por ejemplo:

```
?to timbre
>op char 7
>end
timbre defined
?timbre
?
```

Con esta primitiva es posible enviar caracteres de control al sistema, tal y como ya hemos visto para fijar los colores. Por ejemplo, **char 7** es equivalente a [CTRL][G] según el convenio ASCII, pero al pulsar [CTRL][G] en LOGO el generador de sonido no emite un zumbido, como corresponde al código ASCII; por el contrario, lo único que se consigue es detener la ejecución del programa activo. Puede pulsar [CTRL][G] para comprobarlo. En este contexto [CTRL][G] es equivalente a [ESC]. De todo lo anterior se deduce que debe usarse la primitiva **char** para producir los efectos apropiados de los caracteres de control.

clean

Esta primitiva no lleva ningún parámetro. Su función es borrar la pantalla gráfica sin afectar para nada a la tortuga.

```
?repeat 15 [fd 18 rt 18]
?clean
?
```

co

Esta primitiva reanuda la ejecución de un procedimiento interrumpido tras haber pulsado CTRL Z o por la existencia de una instrucción **pause** dentro del propio procedimiento.

```
?to probar
>pr [uno]
>pause
>pr [dos]
>end
probar defined
?probar
uno
pausing... in probar: [pause]
probar ?co
dos
?
```

Esta instrucción puede resultarnos muy útil, ya que junto con CTRL Z permite averiguar en qué estado se encuentra exactamente el procedimiento que se está ejecutando. Por ejemplo:

```
?viajar
CTRL Z
pausing... in viajar: move
viajar ?tf
[190.603591019247 -40.6615964450271 274 PD 1
TRUE]
viajar ?co
....
```

viajar es un procedimiento que hemos visto con anterioridad.

cos

Esta primitiva proporciona el coseno del ángulo que indica el parámetro medido en grados sexagesimales.

```
?cos 60
0.500000000017049
?
```

Observe que los últimos decimales de la respuesta indican un pequeño error, que

equivale a un error relativo de 3.4097e-11 aproximadamente. Puede calcularse este valor de la siguiente forma:

```
?((cos 60) - .5)/.5
3.40974748436196e-11
?
```

Este tipo de inexactitudes numéricas es bastante frecuente en los ordenadores, puesto que en muchas ocasiones las fracciones numéricas tienen un número infinito de decimales, y desde luego los ordenadores no son infinitos.

count

Esta primitiva utiliza un parámetro, que puede ser una palabra o una lista, y proporciona el número de caracteres que constituyen la palabra o el número de palabras que forman la lista. Convierte los números en su equivalente en caracteres.

```
?count "abcde count [a b c d e] 1/2 count 1/2
5
5
0.5
3
?
```

Al tratar los números como palabras, tanto en esta instrucción como en otras similares, las respuestas resultan a veces un poco extrañas, como por ejemplo:

```
?100000000000000000000000000
1.e+26
?count 100000000000000000000000000
6
?count "100000000000000000000000000
27
?
```

fenómeno examinado con más detalle en la Guía.

Esta primitiva, junto con **item**, permite crear la instrucción **last** que aparece en algunas versiones de LOGO.

```
?to last :listadatos
>op item count :listadatos :listadatos
>end
last defined
?
```

cs

Esta primitiva no emplea ningún parámetro. Su función es borrar la pantalla gráfica colocando a la tortuga en el centro de la misma y apuntando en sentido vertical hacia arriba.

```
?repeat 15 [fd 18 rt 18]
?cs
?
```

ct

Esta primitiva tampoco lleva parámetros. Su función es borrar la pantalla de textos y colocar el cursor en la esquina superior izquierda de la misma. En caso de que nos encontremos en el modo de pantalla compartida, esta primitiva sólo borra la parte inferior de la misma.

```
?pr [no veras esto] ct
?
```

dir

Esta instrucción ofrece una lista de los ficheros de LOGO grabados en el disco que contenga la unidad en este momento; para ello selecciona los ficheros que contengan la extensión **.LOG**. Observe que la extensión del nombre no aparece en la lista de ficheros.

```
?dir
[01 02 03 MATRIZ1 MATRIZ2]
?
```

Todos los nombres de los ficheros aparecen en mayúsculas, pero LOGO acepta tanto **load "matriz1** como **load "MATRIZ1**. Hay que tener cuidado para no confundir esta primitiva con la instrucción del mismo nombre de CP/M.

dot

Esta primitiva toma como parámetro una lista que contenga dos coordenadas y dibuja un punto en el lugar que definen ambas.

```
?dot [30 40]
?dot se 50 60
?
```

Para LOGO **se 50 60** es equivalente a **[50 60]**.

ed

Esta primitiva llama al editor. Para ello requiere un parámetro que puede ser una palabra, (el nombre de un procedimiento o de una variable), o una lista de palabras, (un conjunto de nombres de procedimientos y de variables). En caso de que el mismo nombre se refiera tanto a un procedimiento como a una variable, el editor presenta ambos. Normalmente, si no se incluye ningún parámetro a continuación de **ed**, el editor ofrece una pantalla vacía. Para salir del editor se pulsa la tecla COPY, y para detener la edición ESC.

```
?make "j 0
?to j
>end
j defined
?ed "j
--------
to j
end
make "j 0
--------
```

Si se detecta un error al ejecutar un procedimiento, es posible entrar en el editor escribiendo **ed**. En este caso LOGO coloca el cùrsor en el lugar donde ha detectado el error. En algunos casos el editor confunde el procedimiento, por ejemplo si se produce un error en un procedimiento **a** que a su vez llama al procedimiento **b**, es posible que LOGO edite este último, en lugar de editar **a**. Es frecuente que esto ocurra cuando **a** llama a **b** empleando un parámetro inadecuado.

emtyp

Esta primitiva toma un parámetro e indica si se trata de un conjunto de espacios, es decir, de una palabra "vacía". LOGO no considera los números como palabras vacías.

```
?emptyp "a emptyp [a] emptyp 1
FALSE
FALSE
FALSE
emptyp "emptyp [ ] emptyp 0
TRUE
TRUE
FALSE
?
```

end

Da por terminada la definición de un procedimiento.

```
?to j
>local "j
>pprop "j "j "j
>end
j defined
?
```

Esta primitiva debe encontrarse en una línea separada de las demás; en caso contrario LOGO no la reconoce como final de definición de un procedimiento. Cada procedimiento debe tener un **end** y sólo uno.

ent

Este procedimiento toma como parámetro una lista y define una envolvente de tono con los elementos de la misma. Es equivalente a la instrucción **ent** del BASIC de Locomotive. El primer elemento de la lista es el número de la envolvente, y los restantes definen las diversas secciones de la misma.

```
?make "numero_de_envolvente 1
?make "seccion_envolvente [100 2 20]
?ent se :numero_de_envolvente :seccion_envolvente
?
```

La última línea es equivalente a **ent [1 100 2 20]**. Por último, esta instrucción no admite elementos negativos en la lista.

env

Esta instrucción también toma como parámetro una lista. Su función es definir una envolvente de volumen (véase **ent**).

```
?env se :numero_de_envolvente :seccion_envolvente
?
```

Al igual que la anterior, tampoco admite elementos negativos en la lista.

er

Esta primitiva emplea un parámetro que puede ser una palabra o una lista. Su función es borrar el procedimiento o procedimientos que indica el parámetro, dejando libre al mismo tiempo el área de trabajo correspondiente.

```
?er "j
?er [un dos]
?
```

Si no existe alguno de los nombres, LOGO no llega a realizar la acción de borrado.

```
?to j
>end
j defined
?j
?er [j k]
er doesn't like k as input
?j
?er [j]
?j
I don't know how to j
?
```

ern

Esta primitiva emplea un parámetro que puede ser una palabra o una lista. Su función es borrar la variable o variables que indica el parámetro, dejando libre al mismo tiempo el área de trabajo correspondiente.

```
?ern "j
?ern [uno dos]
?
```

Si no existe alguno de los nombres, LOGO no llega a realizar la acción de borrado.

```
?make "j 0
?ern [k j]
ern doesn't like k as input
?:j
0
?ern [j]
?:j
j has no value
?
```

ERRACT

Esta primitiva tiene por objeto modificar lo que ocurre cuando se produce un error. Habitualmente el valor de **ERRACT** es **FALSE**. Por ejemplo:

```
?to bo
>boob
>end
bo defined
?to boob
>p
>pr [aqui]
>end
boob defined
?bo
I don't know how to p in boob: p
?make "ERRACT "TRUE
?bo
pausing... in boob: p
boob ?po "boob
to boob
p
pr [aqui]
end

boob ?co
?
```

La primitiva **ERRACT** se encuentra estrechamente relacionada con **pause,** CTRL Z y **co**. Este sistema facilita bastante la tarea de eliminar errores en programas complejos.

error

La palabra **error** aparece en dos casos distintos: como etiqueta de la orden **catch** (y del **throw** correspondiente) o bien como **throw "error** (y por consiguiente como **catch "error**) figurando sin parámetros como si se tratase de una instrucción independiente. En este último caso, **error** produce una lista en la que se describe detalladamente el error que se ha producido.

```
?to b
>catch "error [boob]
>end
b defined
```

```
?b
pausing... in boob: p
boob ?co
?to c
>catch "error [boob]
>op error
>end
c defined
?c
pausing... in boob: p
boob ?co
[35 [I don't know how to p] boob [p] catch p]
?
```

Este ejemplo indica que el error 35 es del tipo "**I don't know how to...**". El procedimiento **b** incluye las instrucciones **catch** y **throw**, pero no produce ningún informe del tipo de error; por el contrario, el procedimiento **c** produce un informe del error por medio de **op error**.

FALSE

Se trata de un valor que indica si la proposición anterior es falsa.

```
?2 = 3
FALSE
?2 = 2
TRUE
?
```

FALSE es el contrario de **TRUE**.

fd

Abreviatura de "forward". Esta primitiva hace que la tortuga avance un número de unidades igual al número que la acompaña. Si este número es negativo, la tortuga retrocede en lugar de avanzar.

```
?fd 20
?fd -20
?
```

fence

Esta primitiva no emplea ningún parámetro. Su función es impedir que la tortuga se salga de los límites de la pantalla. Véanse también **wrap** y **window**.

```
?cs
?fd 250
?fence
Turtle out of bounds
?cs
?fence
?fd 250
Turtle out of bounds
?
```

Esta modalidad es mucho menos frecuente que **wrap** o **window**.

first

Esta primitiva da como resultado el primer elemento del objeto introducido en forma de parámetro.

```
?first "abc first [a b c] first 12345
a
a
1
?
```

Si el parámetro es una lista con sublista, **first** actúa de la siguiente forma:

```
?first [[ab c] [d e f] [g h i]]
[ab c]
?first first [[ab c] [d e f] [g h i]]
ab
?first first first [[ab c] [d e f] [g h i]]
a
?
```

fput

Esta primitiva crea una lista añadiendo el objeto que indica el primer parámetro al segundo.

```
?fput "a "bc
abc
?fput "a [bc]
[a bc]
?fput [a] "bc
fput doesn't like bc as input
?fput [a] [bc]
[[a] bc]
?
```

Observe que el primer objeto conserva su carácter, en el sentido de que las listas siguen siendo listas y las palabras siguen siendo palabras. En caso de que el primer objeto sea una lista, el segundo también debe serlo.

fs

Esta primitiva no emplea ningún parámetro. Su función es hacer que toda la pantalla se convierta en pantalla gráfica.

```
?fs
```

Si se dedica la parte inferior de la pantalla a pantalla de textos, la tortuga puede introducirse en esta zona, con lo que desaparece. Con la primitiva **fs** se evita esta desaparición sin que ello afecte a los dibujos que la tortuga hubiese realizado con anterioridad.

glist

Esta primitiva emplea un solo parámetro, y nos proporciona una lista de todos los objetos contenidos en el área de trabajo que tengan la propiedad que indica el parámetro.

```
?glist ".APV
[joyval j ERRACT a]
```

Esto nos permite obtener una lista de todas las variables, procedimientos, primitivas, etcétera, y resulta muy útil para realizar bases de datos.

go

Esta instrucción sólo se puede emplear dentro de un procedimiento y realmente no es recomendable, ya que puede provocar muchas confusiones innecesarias. La ins-

trucción emplea un parámetro, que es una etiqueta, y pasa el control del procedimiento a la línea en que se ha declarado la misma.

```
?to programa_malo
>label "tonto
>pr [es una bobada]
>go "tonto
>end
programa_malo defined
?programa_malo
es una bobada
es una bobada
es una bobada
es una bobada
Stopped! in programa_malo: [pr [es una bobada]]
?
```

Una vez más le recomiendo que no emplee esta primitiva; puede crear muchas confusiones.

gprop

Esta instrucción emplea dos parámetros, el nombre de un objeto y el nombre de una propiedad determinada, y da el valor de la propiedad correspondiente para el objeto en cuestión.

```
?gprop "prueba ".DEF
[[] [local "b] [make "b 9] [op gprop "b ".
APV]]
?
```

Esta primitiva se emplea mucho a la hora de realizar bases de datos y, por otra parte, representa un método bastante sofisticado de controlar los modos de ejecución (véanse los procedimientos para matrices del capítulo 11).

ht

Esta orden no emplea ningún parámetro. Su función es hacer desaparecer la tortuga de pantalla, con lo cual LOGO realiza los dibujos a mayor velocidad.

```
?st
?repeat 20 [fd 18 rt 18]
?cs ht
?repeat 20 [fd 18 rt 18]
?st
?
```

En general es aconsejable emplear esta primitiva a la hora de realizar dibujos, a no ser que haya alguna razón especial para estudiar los movimientos de la tortuga. Dentro de un procedimiento también es posible hacer aparecer y desaparecer la tortuga por medio de CTRL Z.

```
?st ramo 100 5
CTRL Z
pausing... in ramo: ramo
ramo ?ht
ramo ?co
?
```

Estas instrucciones hacen desaparecer la tortuga en mitad del procedimiento **arbol** definido en el capítulo 8.

if

Esta primitiva emplea dos o tres parámetros. El primero de ellos es una proposición que analiza el sistema; el segundo es una lista de instrucciones para ejecutar en caso de que la proposición anterior sea cierta y, por último, el tercer parámetro (que es opcional) es otra lista de instrucciones para ejecutar si la proposición es falsa. En caso de que no exista el tercer parámetro y de que la proposición sea falsa, LOGO salta a la siguiente línea.

```
?if 2 = 2 [pr "verdadero] [pr "falso]
verdadero
?if 2 = 3 [pr "verdadero] [pr "falso]
falso
?if 2 = 2 [pr [verdadero otra vez]]
verdadero otra vez
?if 2 = 3 [pr "seguro?]
?
```

int

Esta instrucción elimina la parte fraccionaria de cualquier número que se introduzca como parámetro.

```
?int 1.2
1
?int -1.2
-1
?
```

Observe que, a diferencia de lo que ocurre en otros lenguajes de programación, el resultado de **int −11.2** no es **−2**, sino **−1**. En LOGO los números negativos tienen el mismo tratamiento que los positivos.

item

Esta primitiva emplea dos parámetros. Su resultado es el elemento del objeto que representa el segundo parámetro y que ocupa el lugar indicado por el primero. El segundo parámetro puede ser tanto una palabra como una lista.

```
?item 3 "abcd item 3 [a b c d] 1/2 item 3 1/2
c
c
0.5
5
?
```

En caso de que el segundo parámetro sea una lista, hay que tener en cuenta que puede estar compuesta por sublistas:

```
?item 2 [a [b cd] e]
[b cd]
?item 2 item 2 [a [b cd] e]
cd
?item 2 item 2 [a [b cd] e]
d
?
```

Se puede emplear esta primitiva, junto con **count**, para crear la primitiva **last** con que cuentan otras versiones de LOGO:

```
?to last :listadatos
>op item count :listadatos :listadatos
>end
last defined
?
```

keyp

Esta primitiva no emplea ningún parámetro. Su resultado es **TRUE** o **FALSE**, según se haya pulsado una tecla que luego podrá ser leída por medio de **rc**, **rl** o **rq**.

```
?repeat 5 [pr keyp repeat 100 [ ]]
FALSE
FALSE
TRUE
TRUE
TRUE
?[
```

En el ejemplo anterior se ha empleado un bucle de retraso (**repeat 100 [ ]**) aunque también podría haberse empleado la primitiva **wait**. Se comprueba cinco veces el teclado y, una vez que es pulsada la tecla, la respuesta a **keyp** pasa a ser **TRUE**. Hemos pulsado la tecla correspondiente al corchete, por eso una vez que el ordenador ha ejecutado el proceso imprime un corchete.

label

Esta primitiva emplea un parámetro, que no es otra cosa que la propia etiqueta que será empleada más tarde para identificar esa línea. Se usa junto con la primitiva **go**.

```
?to programa_malo
>label "tonto
>pr [es una bobada]
>go "tonto
>end
programa_malo defined
?programa_malo
es una bobada
es una bobada
es una bobada
Stopped! in programa_malo: [pr [es un bobada]]
?
```

list

Esta primitiva emplea un número variable de parámetros, formando una lista con todos ellos. La lista resultante conserva el carácter de los objetos empleados como parámetros, es decir, siguen siendo listas o palabras. En caso de que se emplee un solo parámetro o más de dos, es necesario encerrar tanto la primitiva como sus parámetros entre paréntesis.

```
?list "a "b
[a b]
?(list "a "b "c)
[a b c]
```

```
?list "a [b]
[a [b]]
?list [a] "b
[[a] b]
?list [a] [b]
[[a] [b]]
?list [a] [[b]]
[[a] [[b]]]
?
```

Es interesante comparar esta primitiva con **se**.

load

Esta primitiva requiere un parámetro que debe ser un nombre de fichero. Su función es cargar el fichero correspondiente desde el disco en la memoria del ordenador. Éste almacena la información en el fichero en forma de caracteres ASCII, lo que quiere decir que se trata de ficheros de textos, que pueden verse en pantalla desde CP/M por medio de la orden **type** y que pueden modificarse por medio de **ed**. Es muy importante no confundir estas órdenes de CP/M con las órdenes de LOGO que llevan el mismo nombre.

```
?dir
[MATRICES PROLOGO2 PROLOGO1 CAP1 CAP2
ESTADIST]
?load "matrices
bucle defined
matrop defined
matropx defined
?load "estaDIST
abs defined
media defined
mediacuad defined
regres defined
estadist defined
trans defined
?load [matrices estadist]
load doesn't like [matrices estadist] as
input
?
```

El parámetro de esta primitiva ha de ser una palabra, no una lista, y es indiferente que esté escrita en mayúsculas, en minúsculas o en cualquier combinación de ambas. Los procedimientos que aparecen con la palabra "defined" a continuación son los

procedimientos que había en el área de trabajo cuando se grabó el fichero. También están incluidos, aunque no aparezcan en la lista, todos los objetos con propiedades distintas de **.PRM**. Por ejemplo, para continuar con el supuesto anterior:

```
?glist ".APV
[9 8 7 6 5 4 3 2 1 0 b a]
?
```

donde los números no son otra cosa que los objetos empleados para almacenar los resultados del generador de números aleatorios. La Guía incluye más detalles.

local

Esta primitiva se emplea solamente dentro de la definición de un procedimiento. Su función es definir varios objetos, cuyos nombres acepta como parámetros, y restringir su empleo al procedimiento en cuestión:

```
?to prueba_local
>local "a
>make "a [***]
>show :a
>end
prueba_local defined
?make "a 3
?:a
3
?prueba_local
[* * *]
?
```

Esta primitiva es fundamental a la hora de crear sistemas complejos. En general, es aconsejable emplear variables globales lo menos posible, pues de lo contrario consumiremos rápidamente el área de trabajo con bagatelas que realmente no sirven para nada.

lt

Esta primitiva hace que la tortuga gire en sentido contrario al de las agujas del reloj el ángulo introducido en forma de parámetro. En caso de que éste sea negativo la tortuga gira en el sentido de las agujas del reloj.

```
?lt 50
?lt -50
?
```

make

Esta primitiva emplea dos parámetros y asigna el valor del segundo al objeto cuyo nombre es el primer parámetro. Es equivalente a **pprop "nombre ".APV :content.**

```
?make "a 0
?:a
0
?
```

nodes

Esta primitiva no emplea ningún parámetro. Su función es indicar la memoria o área de trabajo que queda libre en cada momento. LOGO consume rápidamente el espacio de memoria con que cuenta, así que con cierta frecuencia es preciso determinar qué contenidos del área de trabajo no son necesarios para reorganizar la misma. Para realizar esta operación se emplea la orden **recycle.**

```
?nodes
526
?recycle
?nodes
1224
?
```

not

Esta primitiva toma un parámetro, que puede ser **TRUE** o **FALSE**, e invierte su valor.

```
?not "TRUE
FALSE
?not "FALSE
TRUE
?
```

op

Esta primitiva toma el valor introducido como parámetro y lo envía como resultado al programa que la llamó, dando por terminada la ejecución del procedimiento en que está incluida. En otros lenguajes de programación hay una función que se encarga de realizar esta acción.

```
?to ensayo
>if rq = "s [op "si]
>pr [no]
>op [esto es una salida]
>pr [linea de comprobacion]
>end
ensayo defined
?ensayo
n
no
[esto es una salida]
?ensayo
s
si
?
```

paddle

Esta primitiva proporciona un valor que representa la orientación del joystick. Si la palanca del joystick se encuentra en su posición de reposo, el valor es 255; en el resto de los casos la orientación la da el valor resultante multiplicado por 45 grados.

```
?repeat 10 [pr paddle 0]
255
255
1
2
255
6
0
3
255
?
```

Como consecuencia de la forma en que el joystick emula al teclado, su uso puede hacer aparecer caracteres en pantalla.

pal

Esta primitiva toma el valor de un parámetro, y responde con una lista de los números que representan las proporciones en que están mezclados los colores básicos para componer la tinta correspondiente al parámetro.

```
?pal 2
[0 2 0]
?
```

Véase **setpal**.

pause

Esta primitiva no emplea ningún parámetro. Se utiliza para detener temporalmente la ejecución de un procedimiento. Para reanudar la ejecución del mismo no hay más que escribir **co**.

```
?to prueba
>pr [uno]
>pause
>pr [dos]
>end
prueba defined
?prueba
uno
pausing... in prueba: [pause]
prueba ?co
dos
?
```

Véase también **co**.

pd

Esta primitiva no emplea ningún parámetro. Su función es bajar la pluma de la tortuga, es decir, hacer que ésta dibuje al moverse.

```
?cs st pd fd 50
?pu fd 50
?pd fd 50
?
```

Puede decirse que esta orden es la inversa de **pu**.

pe

Esta primitiva tampoco emplea parámetros. Su objetivo es hacer que la pluma de

la tortuga tome el color del fondo, con lo que la tortuga va borrando lo que encuentra en su camino al moverse.

```
?setpc 1
?cs tt "XXX
?pe tt "XXX
?
```

plist

Esta primitiva da la lista de propiedades de cualquier objeto que sea introducido como parámetro de la misma.

```
?plist "enrique
[era_padre_de [enrique_VII] padre_de [eduardo
isabel_II] era_madre_de [isabel_I] era [varon]]
?
```

Para su empleo en una base de datos, véase la Guía.

po

Esta primitiva toma un parámetro; en caso de que éste sea una palabra da la definición del procedimiento del mismo nombre. En caso de que sea una lista imprime en pantalla la definición de cada uno de los procedimientos que la componen.

```
?po "yaprop
to yaprop :elem :enlista
if 0 = count :enlista [op "FALSE]
if :elem = first :enlista [op "TRUE] [op
yaprop :elem bf :enlista]
end
?po [yaprop add]
to yaprop :elem :enlista
if 0 = count :enlista [op "FALSE]
if :elem = first :enlista [op "TRUE] [op
yaprop :elem bf :enlista]
end
to add: suj :rel :obj
adprop :suj :rel :obj
adprop :obj word :rel "era_ :suj
end
?
```

pots

Esta primitiva tampoco emplea ningún parámetro. Su función es escribir la primera línea de todos los procedimientos contenidos en el área de trabajo.

```
?pots
to add :suj :rel :obj
to adprop :nom :prop :conten
to yaprop :elem :enlista
to listacon :prop
to conprop :prop :enlista

to condos :l1 :l2
to soloparte
to soloaux :en
?
```

pprop

Esta primitiva emplea tres parámetros: el nombre de un objeto, el nombre de una propiedad y el contenido de esa propiedad para el objeto.

```
?:a
a has no value
?pprop "a ".APV 0
?:a
0
?
```

Para su empleo en una base de datos, véase la Guía.

pr

Esta primitiva acepta uno o varios parámetros y los escribe en la pantalla de textos. Cuando son empleados varios parámetros es necesario encerrarlos, junto con la primitiva, entre paréntesis. La diferencia entre **pr** y **type** consiste en que la primera introduce un retorno de carro después de cada parámetro, mientras que la segunda no. Tanto **pr** como **type** omiten los corchetes exteriores de las listas; por el contrario, **show** imprime en pantalla los corchetes, pero no admite nada más que un parámetro.

```
?pr "a pr "b
a
b
```

```
?show "a show "b
a
b
?type "a type "b
ab
?pr [a] pr [b]
a
b
?show [a] show [b]
[a]
[b]
?type [a] type [b]
ab
?(pr "a "b) (pr "c "d)
a b
c d

?(show "a "b)
a
I don't know what to do with "b
?(type "a "b) (type "c "d)
a bc d
?
```

pu

Esta primitiva no emplea parámetros. Su función es levantar la pluma de la tortuga, lo que impide que ésta dibuje según se mueve.

```
?cs st pu fd 50
?pd fd 50
?pu fd 50
?
```

Podemos considerar que esta orden es la opuesta de **pd**.

px

Esta orden acepta un parámetro, y hace que la tortuga cambie el color de los pixels que se encuentra por su color complementario inverso. En pocas palabras esto quiere decir que la tortuga toma los dos componentes del color y combina los dos números de la pluma, determinando el valor XOR (o exclusivo) de ambos.

```
?setpal 0 [0 0 0] setpal 1 [2 0 0] setpal 2
[0 2 0] setpal 3 [0 0 2]
?cs px
?setpc 1 tt "XXX
?setpc 2 tt "XXX
?setpc 3 tt "XXX
?setpc 0 tt "XXX
?
```

Por ejemplo: el valor binario de 1 es 01, mientras que el valor binario de 2 es 10; el valor 01 XOR 10 es 11, es decir, se obtiene el color correspondiente a la pluma número 3. Es muy interesante probar todas las combinaciones posibles y sacar una tabla comparativa.

random

Esta primitiva proporciona un número aleatorio comprendido entre cero y el valor inmediatamente inferior al parámetro. Al emplear **random** por primera vez después de inicializar LOGO, se obtiene siempre el mismo valor, así que es conveniente echar un vistazo al procedimiento **randomize** empleado en los capítulos 5 y 9.

```
?random 1000
53
?random 1000
94
?random 1000
583
?
```

rc

Esta primitiva no emplea ningún parámetro. Su resultado es el carácter correspondiente a la primera tecla pulsada. Esta orden siempre espera hasta que la tecla haya sido pulsada. Al emplear **rc** dentro de un procedimiento es necesario asignar o comparar su resultado.

```
?if rc = "s [pr "si] [pr "no]
no
?
```

recycle

Esta primitiva no emplea ningún parámetro. Es la encargada de reorganizar el área de trabajo para liberar memoria.

```
?nodes
526
?recycle
?nodes
1224
?
```

Véase también **nodes**.

REDEFP

Se trata de una característica propia del sistema que debe utilizarse con sumo cuidado. Cuando se atribuye el valor **TRUE** a **REDEFP**, el sistema permite volver a definir las primitivas, lo cual puede ser muy útil en algunos casos pero no en otros. Antes de pasar al siguiente ejemplo, grabe en disco el área de trabajo.

```
?to fd :en
fd is a primitive
?make "REDEFP "TRUE
?to fd :en
>end
fd defined
?fd 100
?
```

El ordenador parece no responder a la última instrucción, y realmente no hay forma de salir de esta situación. La tortuga no se puede mover.

release

Esta primitiva acepta un solo parámetro, y es la encargada de liberar los canales de sonido que se hayan retenido en una instrucción **sound**.

```
?release 7
?
```

Este ejemplo libera todos los canales. El parámetro de la instrucción se obtiene sumando los siguientes valores: 1 para el canal A, 2 para el B y 4 para el C.

remrop

Esta primitiva emplea dos parámetros; el primero es el nombre de un objeto y el segundo es una propiedad. Su función es retirar la propiedad del objeto correspondiente.

```
?make "a 0
?plist "a
[.APV 0]
?:a
0
?remprop "a ".APV
?plist "a
[ ]
?:a
a has no value
?
```

También resulta muy útil a la hora de trabajar con bases de datos.

repeat

Esta primitiva también emplea dos parámetros: un número y una lista de instrucciones. El ordenador repite la lista de instrucciones tantas veces como indique el primer parámetro.

```
?repeat 5 [pr "*]
*
*
*
*
*
?
```

rl

Esta primitiva no emplea parámetros. Su función es producir una lista con la información escrita hasta que se produce un retorno de carro, es decir, hasta que se pulsa la tecla INTRO. Si **rl** es usada dentro de un procedimiento, será necesario comparar o asignar su resultado.

```
?if first rl = "s [pr "si] [pr "no]
no
```

```
?show rl
[una lista de palabras]
?show rq
una lista de palabras
?
```

rq

No emplea ningún parámetro. Produce como resultado una palabra (una secuencia de caracteres) con la información escrita hasta que se produce un retorno de carro, es decir, hasta que se pulsa la tecla INTRO. Si se usa **rq** en un procedimiento, su resultado debe ser asignado o comparado.

```
?if first rq = "s [pr "si] [pr "no]
no
?show rl
[una lista de palabras]
?show rq
una lista de palabras
?
```

rt

Esta primitiva hace que la tortuga gire en el sentido de las agujas del reloj el ángulo introducido en forma de parámetro. En caso de que éste sea negativo, la tortuga gira en sentido contrario al de las agujas del reloj.

```
?rt 50
?rt -50
?
```

run

Esta orden se encarga de ejecutar una lista de instrucciones que admite en forma de parámetro.

```
?make "a [fd 100 rt 90]
?run :a
?repeat 1 :a
?
```

De hecho **run** es equivalente a **repeat 1**.

save

Esta primitiva admite como parámetro un nombre de fichero. Su función es grabar en disco, bajo ese nombre, el área de trabajo. LOGO admite como nombre de fichero un grupo de caracteres en mayúsculas, minúsculas o en cualquier combinación de ambas, ya que convierte todas las letras en mayúsculas. Puede solicitar el directorio del disco desde LOGO por medio de la orden **dir**; verá que todos los nombres de ficheros aparecen en mayúsculas.

El nombre del fichero debe ser totalmente nuevo para LOGO, es decir no debe estar en el área de trabajo. La única forma de borrar ficheros es salir a CP/M y emplear la orden **era**.

```
?dir
[ESTADIST PROLOGO1 MATRICES]
?save "MATRICES
File matrices already exists
?save "matriz1
?dir
[MATRIZ1 ESTADIST PROLOGO1 MATRICES]
?
```

se

Esta primitiva admite un número variable de parámetros, formando con todos ellos una lista. No conserva el carácter de los datos, dado que retira los corchetes externos de las listas. En caso de emplear un solo parámetro o más de dos, es necesario encerrar entre paréntesis los parámetros junto con la primitiva.

```
?se "a "b
[a b]
?(se "a "b "c)
[a b c]
?se "a [b]
[a b]
?se [a] "b
[a b]
?se [a] [b]
[a b]
?se [a] [[b]]
[a [b]]
?
```

Es conveniente comparar esta primitiva con **list**.

seth

Esta primitiva admite un parámetro, y se encarga de orientar a la tortuga en el rumbo indicado.

```
?cs tf
[0 0 0 PD 1 TRUE]
?seth 35 tf
[0 0 35 PD 1 TRUE]
?
```

setpal

Esta primitiva emplea dos parámetros: un número del 0 al 3, que representa el color de la pluma, y una lista de tres valores, cada uno de los cuales está comprendido entre 0 y 2, que representa las proporciones de rojo, verde y azul que intervienen en el color correspondiente al primer valor.

```
?cs setpal 0 [0 0 0] [negro negro]
[negro negro]
?setpc 1 setpal 1 [2 0 0] [pluma roja]
[pluma roja]
?setpal 1 [0 2 0] [pluma verde]
[pluma verde]
?setpal 1 [0 0 2] [pluma azul]
[pluma azul]
?setpal 1 [2 2 0] [amarillo brillante]
[amarillo brillante]
?setpal 1 [1 1 0] [amarillo oscuro]
[amarillo oscuro]
?
```

Rojo, verde y azul son los tres colores básicos que componen la luz blanca.

setpc

Esta instrucción hace que la pluma de la tortuga tome el color que corresponde a un número introducido en forma de parámetro. La pluma puede tomar los valores 0, 1, 2 y 3; el número 0 corresponde al color del papel, mientras que los otros tres valores pueden tomar un color diferente que se modifica con la instrucción **setpal**.

```
?setpc 3
?
```

setpos

Esta primitiva toma como parámetros de entrada dos números que representan las coordenadas de un punto, y se encarga de llevar la tortuga hasta la posición indicada. Si se ha bajado previamente la pluma de la tortuga, ésta dibuja una línea según se mueve. La orientación de la tortuga no se ve afectada.

```
?setpos [30 40]
?setpos se 50 60
?
```

Para LOGO **se 50 60** es equivalente a **[50 60]**

setsplit

Esta primitiva permite ajustar por medio de un parámetro el número de líneas de texto de una pantalla compartida. El valor inicial de líneas de texto puede variar entre 1 y 25, pero pruebe con los valores extremos; se producen efectos muy interesantes.

```
?sf
[0 TS 5 WRAP 1]
?setsplit 0
setsplit doesn't like 0 as input
?setsplit 1 cs sf
```

En este caso puede observar que se borra la pantalla y que en la última línea se produce un movimiento muy rápido de caracteres. Al no haber dejado más que una línea para textos, no puede leer la información que envía el sistema.

```
?ts setsplit 26
setsplit doesn't like 26 as input
?setsplit 25
?cs
```

En esta ocasión LOGO borra la pantalla y hace aparecer en la parte superior de la misma la interrogación "?".

```
?sf
[0 SS 25 WRAP 1]
?setsplit 5 sf
[0 SS 5 WRAP 1]
?cs
?
```

Finalmente volvemos a la pantalla compartida habitual.

sf

Esta primitiva no necesita ningún parámetro. Su función es proporcionar una serie de características referentes a la disposición de la pantalla por medio de una lista de cinco elementos, que son los siguientes:

— Color del fondo: siempre es 0.

— Reparto de la pantalla: **TS**, **SS** o **FS**, según el caso.

— Número de líneas para textos que hemos establecido por medio de **setsplit**.

— Carácter de los bordes de la pantalla: **WINDOW**, **WRAP** o **FENCE**.

— Aspecto de la pantalla: siempre es 1.

```
?sf
[0 SS 5 WRAP 1]
?window ts sf
[0 TS 5 WINDOW 1]
?
```

show

Esta primitiva acepta un parámetro y lo imprime en la pantalla de textos. Además conserva los corchetes exteriores de las listas, al contrario de **pr** o de **type** que los eliminan. Otra diferencia con estas dos últimas primitivas es que **show** no admite más que un parámetro. La principal diferencia entre **pr** y **type** consiste en que la primera introduce un retorno de carro después de cada parámetro, mientras que la segunda no lo hace.

```
?pr "a pr "b
a
b
?show "a show "b
a
b
?type "a type "b
ab
?pr [a] pr [b]
a
b
?show [a] show [b]
[a]
[b]
?type [a] type [b]
ab
```

```
?(pr "a "b) (pr "c "d)
a b
c d
?(show "a "b)
a
I don't know what to do with "b
?(type "a "b) (type "c "d)
a bc d
?
```

sin

Esta primitiva proporciona el seno del ángulo que indica el parámetro medido en grados sexagesimales.

```
?sin 30
0.500000000017049
?
```

Observe que los últimos decimales de la respuesta indican un pequeño error, que equivale a un error relativo de 3.4097e−11 aproximadamente. Puede calcularse este valor de la siguiente forma:

```
?((sin 30) - .5)/.5
3.40974748436196e-11
?
```

Este tipo de inexactitudes numéricas es bastante frecuente en los ordenadores, ya que en muchas ocasiones las fracciones numéricas tienen un número infinito de decimales, y desde luego los ordenadores no son infinitos.

ss

Esta primitiva no necesita ningún parámetro. Su función es dividir la pantalla, reservando la parte superior para gráficos y la inferior (5 líneas) para textos.

```
?ss
```

Cuando dedicamos la parte inferior de la pantalla a pantalla de textos, la tortuga puede introducirse en esta zona, y desaparecer. Con la primitiva **fs** se evita esta desaparición sin que ello afecte a los dibujos que la tortuga hubiese realizado con anterioridad.

st

Esta primitiva tampoco emplea parámetros. Su función es hacer que la tortuga reaparezca en pantalla, con lo cual los dibujos se realizan con mayor lentitud.

```
?st
?repeat 20 [fd 18 rt 18]
?cs ht
?repeat 20 [fd 18 rt 18]
?st
?
```

En general es aconsejable emplear la primitiva **ht** a la hora de realizar dibujos, a no ser que haya alguna razón especial para estudiar los movimientos de la tortuga. Dentro de un procedimiento también es posible hacer aparecer y desaparecer la tortuga por medio de CTRL Z.

```
?st ramo 100 5
CTRL Z
pausing... in ramo: ramo
ramo ?ht
ramo ?co
?
```

Las instrucciones anteriores ocultan la tortuga mientras LOGO está ejecutando el procedimiento **arbol**, visto con anterioridad.

stop

Esta primitiva no necesita parámetros. Se emplea para detener la ejecución de un procedimiento. En este caso, el control del programa vuelve al procedimiento de llamada y no se detiene todo el programa, a diferencia de lo que sucede con **throw "TOPLEVEL**.

```
?to uno
>dos
>pr [aqui estamos]
>end
uno defined
?to dos
>if rc = "q [throw "TOPLEVEL] [stop]
>end
dos defined
?uno
<se pulsa q>
```

```
?uno
<se pulsa w>
aqui estamos
?
```

En este ejemplo el procedimiento **uno** llama al **dos**, ejecutando a continuación **pr [aquí estamos]**. El procedimiento **dos** espera hasta que se pulse una tecla y lee el carácter correspondiente. Si éste es una **q**, el procedimiento ejecuta **throw "TOPLEVEL** y se detiene totalmente la ejecución del programa; pero si pulsamos cualquier otra, por ejemplo una **w**, LOGO interrumpe únicamente la ejecución de **dos** y el control pasa al procedimiento que le había llamado, es decir, a **uno**, que continúa ejecutándose.

tf

Esta primitiva no emplea parámetros. Su función es proporcionar información sobre la situación de la tortuga en una lista de seis elementos, que son los siguientes:

— Coordenada x de la tortuga.
— Coordenada y de la tortuga.
— Orientación de la tortuga.
— Posición de la pluma, es decir **PU** o **PD**.
— Número correspondiente a la tinta que está empleando.
— Indicación sobre si la tortuga está visible.

```
?st setpc 1 tf setpc 2 tf
[0 0 0 PD 1 TRUE]
[0 0 0 PD 2 TRUE]
?ht tf setpc 3 tf
[0 0 0 PD 2 FALSE]
[0 0 0 PD 3 FALSE]
?pu tf
[0 0 0 PU 3 FALSE]
?pd repeat 4 [fd 100 rt 90 pr tf pause]
[0 100 90 PD 3 FALSE]
pausing...
[ ] ?co
[100 100 180 PD 3 FALSE]
pausing...
[ ] ?co
[100 0 270 PD 3 FALSE]
pausing...
[ ] ?co
[0 0 0 PD 3 FALSE]
pausing...
[ ] ?co
?
```

En el ejemplo anterior se ha redondeado los valores de las coordenadas a los enteros más próximos, ya que el error cometido es insignificante.

throw

Esta primitiva no emplea más que un parámetro, que es el nombre de una etiqueta que identifica a una orden **catch** previa. Cuando el ordenador llega a la instrucción **throw** salta a la línea siguiente, a la instrucción **catch** correspondiente.

```
?to prueba
>catch "si [otroproc]
>pr [se ha pulsado el boton]
>end
prueba defined
?to otroproc
if buttonp 0 [throw "si]
>otroproc
>end
otroproc defined
?prueba
se ha pulsado el boton
?
```

El procedimiento **prueba** contiene la línea **catch "si [otropoc]**, que se encarga de llamar al procedimiento del mismo nombre. El ordenador guarda en memoria dónde se encuentra la etiqueta **si** del **catch**. Por su parte, **otropoc** se encarga de comprobar si se ha pulsado el botón del joystick y de llamarse a sí mismo en forma recurrente hasta que dicho botón sea pulsado.

Al pulsar el botón el ordenador ejecuta la línea **throw "si**, que devuelve el control a la instrucción **catch "si**. Hay que tener en cuenta que esto sólo sucede si la instrucción **throw** se produce como consecuencia de los procedimientos a los que hemos llamado en la lista de instrucciones de **catch**. Por último, en este caso el control vuelve a la línea siguiente al **catch**, es decir, **pr [...**

to

Esta primitiva admite un número variable de parámetros, el primero de los cuales es el nombre de un procedimiento, e inicia el modo de definición en LOGO. Para ello es preciso que sea el primer comando de la línea. Este hecho es bastante importante, lo que queda demostrado en la siguiente definición:

```
?to to
>pr [que raro]
>end
to defined
?plist "to
[.DEF [[ ] [pr [que raro]]]]
?to
isn't a name or procedure
?pr [LOGO equivocado] to
LOGO equivocado
que raro
?
```

TOPLEVEL

Se trata de una etiqueta que al emplearse con **throw** lleva el control al primer nivel de LOGO, es decir, a la situación en que aparecen el cursor y la interrogación. Es útil para salir de procedimientos anidados con una cierta complejidad. Ya se ha visto que la primitiva **stop** detiene la ejecución de un procedimiento y el control vuelve al procedimiento precedente sin interrumpir la ejecución de todo el programa.

```
?to uno
>dos
>pr [aqui estamos]
>end
uno defined
?to dos
>if rc = "q [throw "TOPLEVEL] [stop]
>end
dos defined
?uno
<se pulsa q>
?uno
<se pulsa w>
aqui estamos
?
```

En este ejemplo el procedimiento **uno** llama al **dos**, ejecutando a continuación **pr [aquí estamos]**. El procedimiento **dos** espera hasta que se pulsa una tecla y lee el carácter correspondiente. En caso de que éste sea una **q** el procedimiento ejecuta **throw "TOPLEVEL**, con lo que se detiene totalmente la ejecución del programa; pero si se pulsa cualquier otra tecla, por ejemplo una **w**, LOGO interrumpe únicamente la ejecución de **dos** y el control pasa al procedimiento que le había llamado, es decir, a **uno**, que continúa ejecutándose.

TRUE

Se trata de un valor que indica si la proposición anterior es cierta.

```
?2 = 3
FALSE
?2 = 2
TRUE
?
```

TRUE es el contrario de **FALSE**.

ts

Esta primitiva no emplea parámetros. Su función es dedicar toda la pantalla a textos.

```
?cs fs wrap fd 5000 ts
?
```

tt

Esta primitiva acepta como parámetro una palabra y se encarga de escribirla en la pantalla de gráficos a partir de la posición en que se encuentra la tortuga. Normalmente, si existen trazos dibujados por la tortuga, el rótulo se superpone al dibujo. Esto se demuestra en el siguiente ejemplo:

```
?ht cs
?repeat 4 [fd 100 tt "X rt 90]
?cs repeat 4 [fd 100 tt "XX rt 90]
?
```

type

Esta primitiva acepta uno o varios parámetros y los escribe en la pantalla de textos. Si se emplean varios parámetros es necesario encerrarlos junto con la primitiva entre paréntesis. La diferencia entre **pr** y **type** consiste en que la primera introduce un retorno de carro después de cada parámetro, mientras que la segunda no. Tanto **pr** como **type** omiten los corchetes exteriores de las listas; por el contrario, **show** imprime en pantalla los corchetes, pero sólo admite un parámetro.

```
?pr "a pr "b
a
b
?show "a show "b
a
b
?type "a type "b
ab
?pr [a] pr [b]
a
b
?show [a] show [b]
[a]
[b]
?type [a] type [b]
ab

?(pr "a "b) (pr "c "d)
a b
c d
?(show "a "b)
a
I don't know what to do with "b
?(type "a "b) (type "c "d)
a bc d
?
```

wait

Esta primitiva detiene la ejecución del programa durante un número de unidades de tiempo indicado por medio de un parámetro. Cada una de estas unidades equivale a 0.22 segundos. Puede emplearse, por ejemplo, para disminuir la velocidad de dibujo de la tortuga. Veámoslo en el procedimiento **fd_lento**:

```
?to fd_lento :long :ciclos
>repeat :long/5 [fd 5 wait :ciclos]
>fd :long - 5 * int (:long / 5)
>end
fd_lento defined
?st tf fd_lento 102 5 tf
[0 0 0 PD 1 TRUE]
[0 102 0 PD 1 TRUE]
?
```

Es posible modificar el valor del segundo parámetro de este procedimiento para ob-

tener la velocidad de dibujo que se desee. En este ejemplo, las coordenadas de la posición de la tortuga que se obtienen con **tf** han sido redondeadas.

window

Esta primitiva no necesita ningún parámetro. Su función es hacer que la pantalla tome el formato habitual, permitiendo que la tortuga se salga de la pantalla y que permanezca fuera de ella hasta que vuelva a tomar coordenadas en el interior de la misma. Las coordenadas de los límites de la pantalla son −200 a +199 en vertical y −320 a +319 en horizontal.

De alguna forma parece que la tortuga dibuja más despacio cuando se encuentra fuera de la pantalla.

```
?window fs cs lt 45 repeat 4 [fd 150 rt 90]
```

Esta línea dedica toda la pantalla a gráficos para que pueda verse qué es lo que sucede en la parte inferior de la misma, dibujando a continuación un cuadrado inclinado con un vértice fuera de la pantalla. Fíjese que el dibujo de este vértice lleva bastante tiempo.

De los tres modos de dibujo en pantalla, **fence**, **window** y **wrap**, ésta última suele ser la más interesante, aunque para algunas aplicaciones, como puede ser el dibujo de diagramas, es preferible emplear **window**.

```
?window cs st rt 25 fd 1020
?wrap
?cs rt 25 fd 1020
?
```

Al introducir la primera línea, la tortuga gira ligeramente y desaparece por la parte superior de la pantalla. Con la segunda línea se produce una pausa, después de la cual la tortuga vuelve a aparecer por la parte inferior izquierda sin dibujar nada más; fíjese en su posición. La última sentencia borra la pantalla y hace que la tortuga gire ligeramente y empiece a dibujar a bastante velocidad multitud de líneas que permiten apreciar el efecto de **wrap**. Observe que la tortuga se detiene en la misma posición que antes.

word

Esta primitiva admite un número de parámetros variable y forma una palabra con todos ellos. Habitualmente lleva dos parámetros, pero es posible emplear solamente uno o más de dos, encerrando en este caso entre paréntesis la primitiva y sus parámetros.

```
?word "a "B
aB
?word 1 2
12
?word "a [b]
word doesn't like [b] as input
?word "
Not enough inputs to word
?(word "a)
a
?word "respuesta "= 1/4
respuesta=
0.25
?(word "respuesta "= 1/4)
respuesta=0.25
?count (word "respuesta "= 1/4)
14
?
```

Es muy frecuente emplear esta primitiva en aplicaciones que exigen el manejo de muchos caracteres, como es el caso de PROLOGO. En la Guía se ha empleado esta instrucción para hacer que LOGO considere un número como un objeto con contenido.

wordp

Esta primitiva toma un parámetro e indica, por medio de **TRUE** o de **FALSE**, si se trata de una palabra. Hay que tener en cuenta que LOGO considera los números como palabras.

```
?wordp "asdf
TRUE
?wordp 12
TRUE
?wordp 1/4
TRUE
?wordp [x]
FALSE
?
```

wrap

Esta primitiva no necesita ningún parámetro. Se encarga de que la pantalla pase

a la modalidad **wrap**, que hace que la tortuga vuelva a aparecer por un extremo de la pantalla al salirse por el extremo opuesto de la misma.

Este formato es el que más se utiliza, dado que con él conseguimos que la tortuga nunca se salga de la pantalla (exceptuando, claro está, la parte inferior de la misma cuando la dedicamos a textos). Hay que tener en cuenta que el formato inicial no es **wrap**, sino **window**.

```
?window cs st rt 25 fd 1020
?wrap
?cs rt 25 fd 1020
?
```

Al introducir la primera línea, la tortuga gira ligeramente y desaparece por la parte superior de la pantalla. Con la segunda línea se produce una pausa, después de la cual la tortuga vuelve a aparecer por la parte inferior izquierda sin dibujar nada más; fíjese en su posición. La última sentencia borra la pantalla y hace que la tortuga gire ligeramente y empiece a dibujar a bastante velocidad multitud de líneas que permiten apreciar el efecto de **wrap**. Observe que la tortuga se detiene en la misma posición que antes.

Apéndice B

Caracteres de control de Dr LOGO

A continuación se incluye una lista completa de los caracteres de control de la versión de Dr LOGO que emplean los ordenadores Amstrad, junto con teclas o combinaciones de teclas que producen resultados equivalentes.

[CTRL][A] o [CTRL][←]	Mueve el cursor hasta el comienzo de una línea de texto. Modos normal y de edición.
[CTRL][B] o [←]	Mueve el cursor un carácter a la izquierda sin borrar el carácter. Modos normal y de edición.
[CTRL][C] o [COPY]	Sale del modo de edición y acepta la última definición del procedimiento correspondiente.
[CTRL][D] o [CLR]	Borra el carácter sobre el que se encuentra el cursor. Modos normal y de edición.
[CTRL][E] o [CTRL][→]	Mueve el cursor hasta el final de una línea de texto. Modos normal y de edición.
[CTRL][F] o [→]	Mueve el cursor un carácter a la derecha. Modos normal y de edición.
[CTRL][G] o [ESC]	Detiene la ejecución en modo normal y la edición cuando se encuentra en el editor.
[CTRL][H] o [DEL]	Borra el carácter situado a la izquierda del cursor. Modos normal y de edición.
[CTRL][I] o [TAB]	Mueve el cursor a la siguiente tabulación. Modos normal y de edición.
[CTRL][J]	No realiza ninguna acción.
[CTRL][K]	Borra el resto de la línea que se encuentra a la derecha del cursor y lo almacena temporalmente en un tampón. Modos normal y de edición.
[CTRL][L]	Se emplea únicamente en modo de edición para colocar en el centro de la pantalla la línea en curso cuando la definición del procedimiento no cabe en la pantalla.

CTRL M o INTRO	Termina una línea de texto con un retorno de carro. En el modo normal coloca la línea en un tampón temporal del que puede recuperarse por medio de CTRL Y. En el modo de edición no es más que un retorno de carro. Compárese con CTRL O.
CTRL N o ↓	Baja el cursor a la siguiente línea. Sólo se puede emplear en modo de edición.
CTRL O	El cursor sigue en la misma línea, pero el resto de la línea que quedaba a su derecha pasa a la de abajo. Sólo se puede emplear en modo de edición.
CTRL P o ↑	Sube el cursor a la línea anterior. Sólo se puede emplear en modo de edición.
CTRL Q o /	Coloca entre comillas el siguiente carácter, que suele ser un delimitador: [ ]": = <> + /. Con este sistema es posible imprimir en pantalla estos caracteres. Sólo se puede emplear en modo de edición.
CTRL R	Retrocede en el tampón hasta el comienzo del texto. Sólo se puede utilizar en modo de edición.
CTRL S o **ss**	Pasa al modo de pantalla dividida. Sólo se puede utilizar en modo normal.
CTRL T o **ts**	Pasa al modo de pantalla de textos. Sólo se puede utilizar en modo normal.
CTRL U o CTRL ↑	Hace que el cursor ascienda una pantalla completa. Sólo se puede utilizar en modo de edición.
CTRL V o CTRL ↓	Hace que el cursor descienda una pantalla completa. Sólo se puede utilizar en modo de edición.
CTRL W	Detiene el desplazamiento vertical del texto, que continúa al pulsar cualquier otra tecla. Sólo se puede utilizar en modo mormal.
CTRL X	Hace que el cursor avance hasta el final del tampón de edición dejando en pantalla una sola línea. CTRL U permite ver más cantidad de texto. Sólo se puede utilizar en modo de edición.
CTRL Y	En modo de edición, recupera la última línea que hemos introducido en el tampón por medio de CTRL K. En modo normal recupera la última línea que hemos introducido con INTRO o con CTRL M.
CTRL Z o **pause**	Detiene la ejecución de un procedimiento, que puede reanudarse por medio de **co**. Sólo se puede utilizar en modo normal.

Apéndice C

Discos y LOGO

Para grabar en disco varios procedimientos, es necesario conocer ligeramente la forma de funcionamiento de CP/M, aunque sólo sea en lo que se refiere a las funciones de tratamiento de discos. La primera parte de este apéndice explica cómo hacer copias del disco que contiene Dr LOGO; más adelante se verá cómo emplear en LOGO los ficheros en disco y, finalmente, algunas otras funciones de interés.

C.1 Cómo proteger su inversión

Haga una copia del disco que contiene Dr LOGO para poder guardar el original en un lugar seguro. En principio supongo que usted todavía no ha grabado en él ningún procedimiento.

Como de momento no vamos a utilizar Dr LOGO, pulse al mismo tiempo [CTRL][SHIFT] y [ESC], con lo que se encontrará en BASIC. Otra posibilidad es escribir desde LOGO la instrucción **bye** y, una vez en CP/M, escribir **amsdos**. Sea cual sea el método que haya empleado para llegar a BASIC, el sistema debe decir:

Ready

Compruebe qué disco tiene en la unidad. Debe ser el disco que contiene Dr LOGO. Dé la vuelta a este disco de forma que quede mirando hacia arriba la cara con CP/M. Ahora mueva hacia abajo o hacia el centro, según el tipo de disco, la placa de plástico de otro color que aparece junto a una flecha en el ángulo superior izquierdo de la funda de plástico que recubre el disco. Esta pieza no es más que un protector, al igual que las placas que hay en los cassettes, que impide que se pierda el contenido de un disco al grabar algo encima. Haga lo mismo con la otra cara del disco, la que contiene Dr LOGO. Finalmente vuelva a introducir el disco en la unidad con la cara que contiene CP/M hacia arriba.

Ahora escriba la siguiente orden:

¦cpm

En caso de que en la pantalla aparezca:

Drive A: disc missing
Retry, Ignore or Cancel?

quiere decir que ha olvidado volver a introducir el disco. Hágalo ahora y pulse la tecla [R]. Esta vez debe encontrarse ya en CP/M 2.2.

El cursor se encuentra junto a la divisa del sistema, **A>**. Escriba:

A>dir

lo que llenará de información la mitad de la pantalla. Se trata del "directorio", una lista con todos los nombres de los ficheros que contiene el disco. A la izquierda de cada línea figura **A:**, que indica que la información procede de la unidad **A**. Esta unidad es la habitual al entrar en CP/M.

Si ahora escribe:

A>dir b:

obtendrá el directorio del disco que se encuentra en la unidad **B**, o en su defecto:

Drive B: disc missing
Retry, Ignore or Cancel?

Esto quiere decir que ha solicitado un directorio del disco que se encuentra en la unidad **B**, pero CP/M no encuentra ningún disco en esta unidad. Aquéllos que tengan dos unidades de disco deberán introducir un disco en **B**, pero los que no dispongan más que de una tendrán que cancelar la orden y volver a la unidad **A** de la siguiente manera:

A>dir b:
Drive B: disc missing

Retry, Ignore or Cancel?

Bdos Err On B: Select
A>

Para contestar a **Select** escriba la letra **A**, aunque no la verá en pantalla.

Antes de seguir adelante hay que hacer una copia de esta cara del disco. Aunque seguramente ya habrá visto en algún sitio cómo realizar esta tarea, voy a volver a explicarlo aquí. Si se detiene a leer el directorio del disco que contiene CP/M, hallará el fichero **DISCCOPY COM**, que realmente corresponde al fichero **DISCCOPY.COM**. La parte del nombre que precede al punto, **DISCCOPY**, es el nom-

bre principal del fichero, mientras que la parte que sigue al punto, **COM**, es lo que denominamos la extensión.

La extensión **COM** significa que el fichero es de instrucciones, lo que se conoce como un fichero de tipo "command". Esto quiere decir que dicho fichero contiene un programa listo para ser ejecutado; para ello no hay más que escribir el nombre principal del fichero, es decir, **disccopy**. Por supuesto, antes de copiar un disco se necesita otro disco en blanco donde grabar la copia. Lo más probable es que utilice un disco virgen, pero por si acaso compruebe que no tiene nada. Para ello introduzca el disco virgen en la unidad y escriba:

```
A>dir
Drive A: read fail

Retry, Ignore or Cancel?

Bdos Err On A: Bad sector
```

Ahora sustituya el disco en blanco por el de CP/M, pulse [INTRO] y volverá a ver su directorio. Como es lógico también es posible utilizar un disco que no esté en blanco, pero siempre es conveniente pedir un directorio del mismo para ver sus contenidos, evitando de esta forma borrar algo que aún interesa.

Aquéllos que dispongan de dos unidades de disco pueden emplear el programa **copydisc** para realizar la copia, pero también resulta interesante ver cómo se realiza la copia con una sola unidad. Para iniciar la copia introduzca el disco de CP/M y escriba:

```
A>disccopy

DISCCOPY V2.0
Please insert source disc into drive A then press any key:_
```

En este caso el disco origen es el que contiene CP/M y ya se encuentra en la unidad, por tanto no hay más que pulsar [INTRO]. A continuación el ordenador indica que comienza a leer pistas y, una vez que ha leído ocho (de la 0 a la 7), le pide que introduzca el disco destino y que pulse cualquier tecla.

El disco destino es el que había preparado para grabar en él la copia. Lo único que tiene que hacer es sacar de la unidad de disco el que contiene CP/M e introducir el disco en blanco. A continuación el ordenador informa de que ha comenzado a copiar los contenidos en el disco, inicializándolo previamente. CP/M no puede leer datos de un disco en blanco sin inicializarlo antes. Esta tarea consiste en adaptar el disco a las condiciones de trabajo del sistema operativo; para ello existe un programa especial que se puede encontrar en el fichero **FORMAT.COM**. Cuando copie un disco sobre otro que ya haya utilizado anteriormente, **disccopy** no tendrá que inicializarlo, dado que si ya contenía algo es evidente que en algún momento tuvo que ser inicializado.

No obstante, para evitar complicaciones **disccopy** inicializa automáticamente el disco destino cuando es necesario, al igual que **copydisc**. A continuación repita varias veces el proceso de copiar ocho pistas cambiando una y otra vez los dos discos. Cuando llegue al final el sistema indicará:

Copying complete
Do you want to copy another disc (Y/N):_Y
Please insert source disc into drive A then press any key:_

A continuación vamos a copiar el disco de Dr LOGO, así que retire la copia del sistema operativo que acaba de realizar y copie en la otra cara el disco original de Dr LOGO. Una vez que haya terminado de copiar el disco, compruebe que todo marcha bien dando la siguiente instrucción sin sacar de la unidad de disco la copia de Dr LOGO:

DISCCOPY V2.0 finished

A>dir
A: LOGO COM : SETUP COM : AMSDOS COM
A>)

Ahora que tiene una copia de Dr LOGO guarde el disco original en un lugar seguro. En caso de que el disco contenga otros ficheros con la extensión **LOG**, quiere decir que en algún momento había grabado en el disco original algún procedimiento de LOGO. Todos los ficheros con la extensión **LOG** contienen procedimientos, pero en la siguiente sección se tratará un poco más sobre este tema.

Ahora que hemos visto cómo funciona **disccopy** y antes de seguir adelante, observe que cuando el ordenador solicita un disco de sistema con **CP/M system disc ...** puede utilizarse cualquier disco que contenga la información básica de CP/M. En este sentido, la mayor parte de los discos con los que va a trabajar son discos de sistemas. El disco con Dr LOGO es uno de ellos, ya que tiene pistas con información propia del sistema que no aparecen al solicitar un directorio; pero tenga en cuenta que este disco es diferente del disco de CP/M, que además de estas pistas contiene varios ficheros con programas que sí aparecen si solicita un directorio, como por ejemplo **FORMAT.COM**.

C.2 Ficheros de LOGO

Antes de seguir adelante, es preciso que defina algunos procedimientos y los grabe en disco con la primitiva **save**.

Para simplificar un poco las cosas grabe un solo fichero de LOGO con la siguiente instrucción:

?save "primero
?

Sin entrar por el momento en lo que pudiera contener el área de trabajo al grabarla, escriba:

```
?dir
[PRIMERO]
?
```

con lo que obtiene un directorio de los ficheros LOGO que contiene el disco. Desde luego no es una lista completa; fíjese que no aparecen más que los ficheros cuyo nombre tiene la extensión LOG. Además, aunque ha grabado el fichero con el nombre **primero** (en minúsculas), en la lista que contiene el directorio, que de hecho es una lista como cualquier otra, aparece **PRIMERO**, en mayúsculas. Si hubiese grabado también otro fichero, por ejemplo **n**, la lista obtenida sería la siguiente:

```
?dir
[PRIMERO N]
?
```

Observe que es posible trabajar con esta lista como con cualquier otra:

```
?item 2 dir
N
?
```

Esta instrucción ha seleccionado el segundo elemento de la lista.

Salga de LOGO con **bye**, vuelva a CP/M y solicite desde ahí el directorio:

```
A>dir
A: LOGO COM : SETUP COM : AMSDOS COM
A: PRIMERO LOG
A>
```

Observe que el fichero que grabó desde LOGO tiene el nombre **PRIMERO.LOG**. Intente ver qué contiene. Escriba:

```
A>type primero.log
to otromodo :num
type word char 4 char (:num + 32)
end
make "1 2

A>
```

y compruebe que se trata de dos experimentos realizados en LOGO: un procedimiento llamado **otromodo**, que permite cambiar el modo de ejecución (¡cuidado

con él!), y una instrucción **make**, que atribuye al objeto **1** el valor **2**. La orden **type** de CP/M también permite estudiar el contenido de los ficheros de LOGO. ¿Se puede emplear esta instrucción con cualquier tipo de fichero? La respuesta es no. La instrucción **type** sólo funciona correctamente con ficheros codificados en ASCII, es decir, con ficheros guardados como si se tratase de un conjunto de caracteres legibles.

Si emplea **type** con ficheros que no estén codificados en ASCII obtendrá algo totalmente ilegible. Los ficheros **COM**, por ejemplo, no son legibles con **type**. Haga la prueba:

```
A>type logo.com
```

y se perderá irremisiblemente. La única solución es pulsar al mismo tiempo las teclas CTRL, SHIFT y ESC o escribir **amsdos**, si es que puede escribir a ciegas.

Desde LOGO es factible modificar los procedimientos, pero no se puede volver a grabarlos con el mismo nombre que antes. Por ejemplo:

```
?save "primero
File primero already exists
?
```

El ordenador advierte que **primero** ya existe. Este artificio tiene como única finalidad impedir que se pierdan los contenidos de un fichero al grabar algo encima del mismo. Dr LOGO obliga a pasar a CP/M para borrar un fichero o cambiarle el nombre, de forma que no es posible eliminar un fichero por error.

Para manejar ficheros desde CP/M conviene conocer unas cuantas funciones que son muy útiles.

C.3 Otras funciones de interés

Si solicitase un directorio de uno de mis discos de LOGO desde CP/M, obtendría algo así:

```
A>dir
A: LOGO COM : SETUP COM : AMSDOS COM : STAT COM
A: UPS LOG
A>
```

Desde luego, **UPS.LOG** es un fichero de LOGO, pero **STAT.COM** no aparece en el disco original de Dr LOGO. Se trata de un fichero que he copiado desde el disco de CP/M por una razón determinada: es fundamental para trabajar con cualquier disco.

Para pasar este fichero al disco de LOGO, no hay más que introducir el disco de

CP/M en la unidad y ejecutar la orden **filecopy** aplicándola al fichero **STAT.COM**. Aquéllos que cuenten con dos unidades de disco pueden copiar el fichero con **pip**, aunque en este caso no suele ser necesario pasar también el fichero **PIP.COM** al disco de LOGO. Una vez copiado **STAT**, vamos a ver para qué sirve.

En primer lugar escriba:

```
A>stat *.*
  Recs  Bytes  Ext  Acc
     2     1k    1  R/W A:AMSDOS.COM
   256    32k    2  R/W A:LOGO.COM
    61     8k    1  R/W A:SETUP.COM
    61     8k    1  R/W A:STAT.COM
    41     6k    1  R/W A:UPS.LOG
     1     1k    1  R/W A.UPS.LOG
Bytes Remaining On A: 121k

A>
```

De momento no se preocupe por los títulos de las columnas. El primer dato que interesa es la cantidad de bytes libres en el disco, ya que necesita saber si queda espacio para grabar más ficheros. También puede comprobar que todos los ficheros contienen la indicación R/W, gracias a la cual sabrá que puede leer y escribir en cada uno de ellos. Esto quiere decir que es posible escribir encima del fichero **LOGO.COM**. Para evitarlo hay que darle la característica "read only", que impide modificarlo.

Para ello se emplea la siguiente instrucción:

```
A>stat logo.com $r/o

LOGO.COM set to R/O
A>stat *.*
  Recs  Bytes  Ext  Acc
     2     1k    1  R/W A:AMSDOS.COM
   256    32k    2  R/O A:LOGO.COM
    61     8K    1  R/W A:SETUP.COM
    41     6k    1  R/W A:STAT.COM
     1     1k    1  R/W A:UPS.COM
Bytes Remaining On A: 121k

A>
```

Si ha conseguido dar a **LOGO.COM** la característica **R/O**, intente borrarlo con **era**:

```
A>era logo.com
Bdos Err On A: File R/O
A>
```

Ahora ya está mucho más tranquilo, ¿verdad?

Como puede comprobar, la única forma de aprender cómo funciona CP/M consiste en utilizarlo. CP/M es un sistema operativo muy útil y juega un papel importante incluso al trabajar con Dr LOGO. Dos funciones útiles son las siguientes:

— Es posible emplear el editor de CP/M, **ed**, para modificar los ficheros grabados desde Dr LOGO. Hay que hacer notar que, si su ordenador sólo cuenta con una unidad de disco, tendrá que copiar el fichero **ED.COM** en el disco que contiene LOGO. Este sistema permite depurar un poco los ficheros borrando los procedimientos y variables que no vayan a ser utilizados. No se olvide de borrar la copia de seguridad del fichero. Se distingue por la extensión BAK.

— La orden **pip** permite introducir ficheros de LOGO directamente desde CP/M. Por ejemplo:

A>pip trial.log =con:

permite introducir a través del teclado un fichero denominado **trial.log**. No olvide que para finalizar cada línea hay que pulsar [INTRO][CTRL][J], y que para terminar el fichero hay que pulsar [CTRL][Z].

Antes de hacer uso de estas dos funciones es conveniente que adquiera más experiencia con CP/M, especialmente en lo que se refiere a la creación de ficheros LOGO con **pip**.

Apéndice D

Ampliaciones de LOGO

Este apéndice tiene por objeto explicar cómo se puede escribir ampliaciones de LOGO desde código de máquina, y cómo pueden redefinirse las primitivas para obtener otras formas de funcionamiento.

Por ejemplo, suponga que queremos modificar la definición de **fd** de manera que esta primitiva, además de dibujar líneas en pantalla, mueva una tortuga que se encuentre en el suelo. Para modificar esta definición conservando la posibilidad de dibujar en pantalla sin mover la tortuga del suelo, vamos a definir otro procedimiento, al que llamaremos **avanzar**, que actúe exactamente igual que el antiguo **fd**.

La clave la ofrece la propiedad **.PRM**. Todas las primitivas de LOGO tienen la propiedad **.PRM**, siendo el valor de esta propiedad la dirección de memoria donde comienza la rutina en código de máquina que activa la primitiva correspondiente. Por ejemplo:

```
?gprop "fd ".PRM
6365
?pprop "avanzar ".PRM gprop "fd ".PRM
?repeat 4 [avanzar 100 rt 90]
?
```

Hemos atribuido a **avanzar** la acción que antes realizaba **fd**. Esto quiere decir que ahora podemos volver a definir tranquilamente **fd**, porque **avanzar** realiza la misma acción.

Antes de seguir adelante por este camino, hay que hacer un poco de sitio en memoria para almacenar las nuevas rutinas en código de máquina, y además es preciso dar otro valor a la propiedad **.PRM** de **fd**. Esta última tarea es muy sencilla:

```
?pprop "fd ".PRM 48148
?
```

Antes de emplear la nueva versión de **fd**, grabe en disco el área de trabajo como medida de seguridad. Ahora escriba:

```
?fd
```

Las consecuencias pueden ser muchas y muy variadas, pero lo más normal es que pierda el control del ordenador y tenga que pulsar CTRL SHIFT y ESC.

Para ampliar la memoria de forma que admita nuevas rutinas en código de máquina, es necesario volver a CP/M y reducir el espacio que ocupa el propio CP/M por medio del comando **movcpm**. La siguiente orden permite saber cuál es el espacio ocupado por CP/M:

A>movcpm 179 *

pero con la instrucción:

A>movcpm 171 *

el tamaño de CP/M queda reducido dejando libres 2K para rutinas en código de máquina, es decir, (179−171)*256 bytes. Con ello ha bajado el límite superior de CP/M desde la posición .AD32 (44338) hasta la .A532 (42290). Esto quiere decir que las nuevas rutinas en código de máquina deben tener valores de **.PRM** comprendidos entre 42290 y 44338.

Para grabar en disco la nueva configuración de CP/M, escriba:

A>sysgen *

y siga las indicaciones del ordenador.

Antes de entrar en LOGO hay que cargar las rutinas en código de máquina en el espacio reservado para tal fin. Para ello normalmente se emplea un programa de aplicaciones específico, cuya descripción rebasa el ámbito de este apéndice. Hay que advertir que el sistema normal de LOGO cuenta con 2105 nodos, mientras que el sistema con configuración de CP/M reducida en 2K no cuenta más que con 1609 nodos. Hay que compensar el espacio que ocupan las nuevas rutinas en código de máquina con el espacio que queda disponible para los procedimientos normales.

Si se quiere conservar una redefinición, hay que incluirla en un procedimiento que pueda ser grabado como parte del área de trabajo. Por ejemplo:

```
?to redef
>pprop "avanzar ".PRM gprop "fd ".PRM
>pprop "fd ".PRM 0
>end
redef defined
?save "procs
?
```

Para recuperar estas dos nuevas definiciones no hay más que cargar el fichero **procs** y ejecutar el procedimiento **rederf**.

Índices

Índice A

Órdenes y primitivas de LOGO usadas en el texto

A

abs 90, 91
.APV 71-76, 81, 100, 102, 118

B

bf 83, 122
bk 16, 68, 123
buttonp 51, 124
bye 4, 12, 38, 124

C

catch 49, 53, 124
co 126
.contents 30, 31, 71, 74, 119
count 44, 81, 83, 127
cs 9, 128
char 45, 46, 125

D

.DEF 72, 76, 119
dir(LOGO) 4, 28, 128, 174

E

ed 20, 23, 73, 129
edall 77
end 19, 130
ent 61, 130
env 59, 60, 130

F

FALSE 43, 88, 133
fd 7, 9, 13, 24, 31, 133
fence 12, 134
first 83, 134
fput 87, 114, 134
fs 6, 9, 68, 135

G

glist 71, 112, 113, 135
gprop 73, 76, 98, 136

H

ht 14, 136

I

if 43, 52, 83, 90, 137
int 101, 137
item 82, 99, 138, 173

K

keyp 43, 138

L

list 86, 87, 139
load (LOGO) 28, 140
local 51, 52, 57, 77, 90, 141
lput 87, 88
lt 7, 9, 24, 141

M

make 51, 98, 113, 142

N

nodes 27, 142
not 142

O

op 51, 57, 60, 73, 76, 83, 90, 142

P

paddle 51, 143
pd 144
plist 72, 73, 75, 81, 103, 145
po 19, 35, 145
pprop 97, 109, 110, 146
pr 72, 91, 146
.primitivas 74
.PRM 73, 120
pu 68, 147

R

random 38, 78, 80, 148
randomize 39
rc 44, 148
recycle 27, 53, 149
remprop 150
repeat 13, 24, 35, 40, 98, 150
rl 43, 150
rt 7, 24, 151
run 151

S

save (LOGO) 152
se 58, 85-87, 93, 152
setpal 47, 153
setpc 47, 48, 153
sound 55-59
ss 6, 156
st 5, 6, 15, 157
stop 49, 65, 157

T

tf 53, 158
throw 49, 53, 159
to 19, 75, 159
TOPLEVEL 160
TRUE 43, 88, 161
"TRUE 43
ts 6, 31, 161
tt 51, 51, 161

W

window 12, 163
word 46, 47, 76, 78, 163
wordp 88, 164
wrap 11, 12, 22, 42, 164

Y

yaprop 112

! 9

" 20, 32, 34, 82

* 51, 117

+ 84, 117

− 84, 118

/ 81, 120

: 20, 32, 34

= 85, 121

> 121

>(mensaje inductor) 19

?(interrogación) 3, 5, 19

Índice B

Procedimientos definidos en el texto

to a 72
to a :nada 73
to abs :a 90
to add :suj :rel :obj 110
to adprop :nom :prop :conten 110
to altobajo 63
to arbol :l :o 65
to arbol_inc :l :a :o :f1 :f2 68
to arbol_int :l :a :o :f1 :f2 69
to borrainput 44
to borrar :n 78
to bucle :var :in :fin :lista_bucle 99
to col :nom :c 99
to col.viajar? 49
to col.viajarit? 49
to colorborde :valor 46
to colorpapel :valor 47
to colortinta :tinta :color 47
to comb :oper :l1 :l2 86
to combinar :oper :l1 :l2 86
to comprobar :num :rep 77
to condos :l1 :l2 112
to conprop :prop :enlista 112
to copiar :dest :orig 103
to corr :cov :sd1 :sd2 92
to correg :cov :vary :varx :my :mx 93
to covar :l1 :l2 :m1 :m2 90
to crear :nom :apv :dim 103
to dibujoy 51
to edall 77
to elem :nom :f :c 101
to empezar 64
to estadist :listy :listx 94
to estrella :lado :angulo 19
to estrella :lado :angulo :veces 33
to fila :nom :f 99
to girar 50
to invertir :nom 103
to joy 51
to jugar 64
to leerjoy 50
to leertecla 52
to listacon :prop 112
to listar :n 78
to lput :elem :listadatos 88
to mas :x 77
to matrlist :nom 98
to matrop :res :oper :m1 :m2 105
to matropx :res :oper :m1 :m2 106
to media :listadatos 88
to mediacuad :listadatos 89
to mediaprod :l1 :l2 89
to mitad :l :o 67
to mod :num :div 101
to mover 38, 40, 50
to .nombres 74
to paseo_corto 40
to paseo_largo 41
to periodo :nota 57
to pops 77
to .primitivas 74
to raiz2 :num 90
to randomini 78
to randomize 39
to regcoef :cov :var 92
to regconst :my :mx :rcoe 92
to regres :my :mx :varx :cov 93
to rotular 52
to runop :oper :p1 :p2 86
to separar :en 114
to sepaux :nom :en 114
to sigma :l 107
to sigma :listadatos 83, 85
to soloaux :en 113

to soloparte 113
to tam :nom :dim 97
to thing :nombre 73
to to 75
to tortuga 50
to trans :dest :orig 103
to transval :nom 103
to vacilar 42
to varianza :listadatos :mediadatos 89
to viajar 41
to viajar? 43
to yaprop :elem :enlista 111
to #bucle :var :ini :fin :lista_bucle 99

Índice C

Índice general

A

ALTOBAJO, juego 63
amsdos 5
APL 95, 97, 100, 104
Árbol 65
Arbol inclinado 68
Área de trabajo 31
Aritmética, prioridades en 83
ASCII, ficheros 174

B

BAK (extensión) 176
Base de datos,
  creación de una 111
  FAMILIA 111
  grabación de una 113
  recuperación de la 114
Bdos Err 170
Binarios, ejercicios 63
Bucle FOR 99
Bucles controlados 98
Búsqueda de la tortuga 5

C

Canal, estado del 55
Caracteres de control 45
Carga de LOGO 3
Circunferencia 14, 15
Códigos de control 45
Color 45
COM (extensión) 171, 172
Combinación de matrices 104
Comillas 20, 34
Control del movimiento 42
Copia de un disco 170
Copia de seguridad 169
[COPY] 21
Copydisc 171
Correlación,
  coeficiente de 92
  y regresión 92
Corte binario 67
CP/M 4
  carga de 169, 170
  disco de 3
[CTRL]G 18, 42, 45, 167
[CTRL]Q 82, 168
[CTRL]Y 9, 18, 40, 168
Covarianza 90
Cuadrado 9

D

Datos y parámetros 24
Definición, modo de 19
Delimitadores 23
Desviación típica 90
Dibujos con rótulos 52
dir (CP/M) 170
Directorio (LOGO) 173, 174
Directorio (CP/M) 170
Disco origen 171
disccopy 171
Dos puntos 20, 34

E

ent 61
Entrenamiento con la tortuga 7
era 175
Errores 29
  escritura de 6
  localización de los 30
[ESC] 18, 21, 41, 42
Espacio(s)
  como delimitadores 23
  entre palabras 23
Estadística con dos variables 88
Estrella(s)
  fugaces 17
  nacimiento de la 21
  nueva 35
Extensión 172

F

Ficheros
  ASCII 174
  introducidos a través del teclado 176
  de lectura/escritura (R/W) 175
  de sólo lectura (R/O) 175
  de LOGO 172
Filas como sublistas 96
FOR, bucle 99
Fullscreen 7

G

Grabación
  en el área de trabajo 26
  de una base de datos 113

H

Hechos y relaciones 109

I

Índices de una matriz 96
Infijo, operador 84
Inicialización de discos 171
Instrucciones, lista de 14
Intérprete de LOGO 31
Interrogación (?) 3

J

Juego de los teléfonos 25

L

Lista(s)
  combinación de 85
  de instrucciones 14
  suma de los elementos de una 83
Localización de errores 30
LOG (extensión) 172
LOGO
  desde BASIC 5
  carga de 3
  desde CP/M 5
Llamar al procedimiento 34

M

Matriz(ces) 95
  combinación de 104
  elección de un elemento 100
  indexadas 102
  índices de las 96
  como propiedad 96
  como tabla 95
  tamaño de las 97
  transposición de 102
  traspuestas 96, 102
Media 88
  de los cuadrados de los valores 88
  de los productos cruzados 89
Memoria, reorganización de la 27
Módulos, sistema estadístico en 93
Movimiento de la tortuga 49

N

Newton, método de 90
Nombre(s) 20, 71
  de fichero 170

objetos y propiedades 33
y propiedades 71
Números aleatorios, generación de 37

O

Objeto 71
Operador infijo 84
Operador prefijo 84

P

Palabras
y números 81
en la pantalla de gráficos 52
precedida de comillas 34
Pantalla
completa 7
dividida 6
gráfica 6
de textos 6
Parámetro 22
Parámetros y datos 24
Pentágono 17
Pila (stack) 48, 49
pip 175, 176
Polígono 15
Prefijo, operador 84
Primitiva 6, 7
Primitivas con número variable de parámetros 76
Procedimiento 6, 30
Procedimiento recurrente 42
Protección contra escritura 169
Propiedad(dades) 71
la matriz como 96
valores de las 71
Pruebas aleatorias 76

R

Raíz cuadrada 90
Recuperación de la base de datos 114
Recurrencia 37, 42, 49, 78
Recurrencia múltiple 67
Recurrente, procedimiento 42
Regresión
coeficiente de 92
constante de 92
y correlación 92
recta de 92
Relaciones y hechos 109
Release 62
R/O, ficheros 175
R/W, ficheros 175

S

Sistemas estadísticos con dos variables 93
Sound 55
Stack (pila) 48, 49
stat 175
Sublistas 96
Suma de los elementos de una lista 83

T

Teclado, introducción de ficheros a través del 176
Textos y gráficos 52
Textscreen 6
Tono 57
Tortuga
movimiento de la 49
vacilante 40
viajeras 13
type (CP/M) 174

V

Valores
procedentes de un procedimiento 51
de los valores 74
Varianza 89

¦cpm 169

[ ] como delimitadores 23


AMSTRAD
E S P A Ñ A
AVDA. DEL MEDITERRANEO, 9 - 28007 MADRID